z/OS
Integrated Cryptographic Service Facility

Application Programmer’s Guide

SA22-7522-02

z/OS
Integrated Cryptographic Service Facility

Application Programmer's Guide

SA22-7522-02

Note!

Before using this information and the product it supports, be sure to read the general information in the “Notices” on page 453.

Third Edition (March 2002)

This is a major revision of SA22-7522-01.

This edition applies to Version 1 Release 3 of z/OS (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/webqs.html

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

About This Book

This book describes how to use the callable services provided by the Integrated Cryptographic Service Facility (ICSF). The z/OS Cryptographic Services includes these components:

- Integrated Cryptographic Service Facility
- z/OS Open Cryptographic Services Facility (OCSF)

ICSF is a software element of z/OS that works with the hardware cryptographic feature and the Security Server (RACF) to provide secure, high-speed cryptographic services. ICSF provides the application programming interfaces by which applications request the cryptographic services.

Who Should Use This Book

This book is intended for application programmers who:

- Are responsible for writing application programs that use the security application programming interface (API) to access cryptographic functions.
- Want to use ICSF callable services in high-level languages such as C, COBOL, FORTRAN, and PL/I, as well as in assembler.

How To Use This Book

ICSF includes both Data Encryption Standard (DES) and public key cryptography. These are two very different cryptographic systems.

Part 1 focuses on IBM CCA programming. It includes the following chapters:

- Chapter 1, “Introducing Programming for the IBM CCA” describes the programming considerations for using the ICSF DES callable services. It also explains the syntax and parameter definitions used in callable services.
- Chapter 2, “Introducing DES Cryptography and Using DES Callable Services” gives an overview of DES cryptography and provides general guidance information on how the DES callable services use different key types and key forms. It also discusses how to write your own callable services called installation-defined callable services and provides suggestions on what to do if there is a problem.
- Chapter 3, “Introducing PKA Cryptography and Using PKA Callable Services” introduces Public Key Algorithm (PKA) support and describes programming considerations for using the ICSF PKA callable services, such as the PKA key token structure and key management.

Part 2 focuses on CCA callable services and includes the following chapters:

- Chapter 4, “Managing DES Cryptographic Keys” describes the callable services for generating and maintaining cryptographic keys, the random number generate callable service (which generates 8-byte random numbers) and the Secure Sockets Layer (SSL) security protocol. It also presents utilities to build DES tokens and generate and translate control vectors and describes the PKA callable services that support DES key distribution.
- Chapter 5, “Protecting Data” describes the callable services for deciphering ciphertext from one key and enciphering it under another key. It also describes enciphering and deciphering data with encrypted keys and encoding and decoding data with clear keys.

- Chapter 6, “Verifying Data Integrity and Authenticating Messages” describes the callable services for generating and verifying message authentication codes (MACs), generating modification detection codes (MDCs), generating hashes (SHA-1, MD5, RIPEMD-160), and generating and verifying VISA card verification values.
- Chapter 7, “Financial Services” describes the callable services for generating, verifying, and translating personal identification numbers (PINs). It also describes the callable services that support the Secure Electronic Transaction (SET) protocol.
- Chapter 8, “Using Digital Signatures” describes the PKA callable services that support using digital signatures to authenticate messages.
- Chapter 9, “Managing PKA Cryptographic Keys” describes the PKA callable services that generate and manage PKA keys.
- Chapter 10, “Utilities” describes callable services that convert data between EBCDIC and ASCII format, convert between binary strings and character strings, and edit text strings according to ANSI X9.9-4 editing rules.
- Chapter 11, “Trusted Key Entry Workstation Interfaces” describes the PCI interface (PCI) and the Public Key Secure Cable (PKSC) interface that supports Trusted Key Entry (TKE), an optional feature available with ICSF.
- Chapter 12, “Managing Keys According to the ANSI X9.17 Standard” describes the callable services that support the ANSI X9.17 key management standard [1], which defines a process for protecting and exchanging DES keys.

The appendixes include the following information:

- Appendix A, “ICSF and TSS Return and Reason Codes” explains the return and reason codes returned by the callable services.
- Appendix B, “Key Token Formats” describes the formats for DES internal, external, and null key tokens and for PKA public, private external, and private internal key tokens containing either Rivest-Shamir-Adleman (RSA) or Digital Signature Standard (DSS) information. This appendix also describes the PKA null key token.
- Appendix C, “Control Vectors and Changing Control Vectors with the CVT Callable Service” on page 383 contains a table of the default control vector values that are associated with each key type and describes the control information for testing control vectors, mask array preparation, selecting the key-half processing mode, and an example of Control Vector Translate.
- Appendix D, “Coding Examples” provides examples for COBOL, assembler, and PL/1.
- Appendix E, “Using ICSF with BSAFE” explains how to access ICSF services from applications written using RSA’s BSAFE cryptographic toolkit.
- Appendix F, “Cryptographic Algorithms and Processes” on page 419 describes the PIN formats and algorithms, cipher processing and segmenting rules, multiple encipherment and decipherment and their equations, the PKA92 encryption process, partial notarization of an ANSI key-encrypting key (AKEK), and the algorithm for transforming a Commercial Data Masking Facility (CDMF) key.
- Appendix G, “EBCDIC and ASCII Default Conversion Tables” presents EBCDIC to ASCII and ASCII to EBCDIC conversion tables.
- Appendix H, “Access Control Points and Callable Services” lists which access control points correspond to which callable services.
- Notices contains notices, programming interface information, and trademarks.

[1] ANSI X9.17-1985: Financial Institution Key Management (Wholesale)

Where To Find More Information

For information about the referenced ICSF books, see Figure 1 on page xxiv.

Other books referenced in this book are:

- *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*, SC40-1675
- *z/OS MVS Programming: Callable Services for HLL*, SA22-7613
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*, SA22-7611
- *BSAFE User's Manual*
- *BSAFE Library Reference Manual*

Related Publications

- *z/OS ICSF TKE Workstation User's Guide 2000*, SA22-7524
- *IBM Transaction Security System: General Information Manual and Planning Guide*, GA34-2137
- *IBM Transaction Security System: Concepts and Programming Guide: Volume I, Access Controls and DES Cryptography*, GC31-3937
- *IBM Transaction Security System: Concepts and Programming Guide: Volume II, Public-Key Cryptography*, GC31-2889
- *IBM Transaction Security System: Basic CCA Cryptographic Services*, SA34-2362
- *IBM Distributed Key Management System, Installation and Customization Guide*, GG24-4406

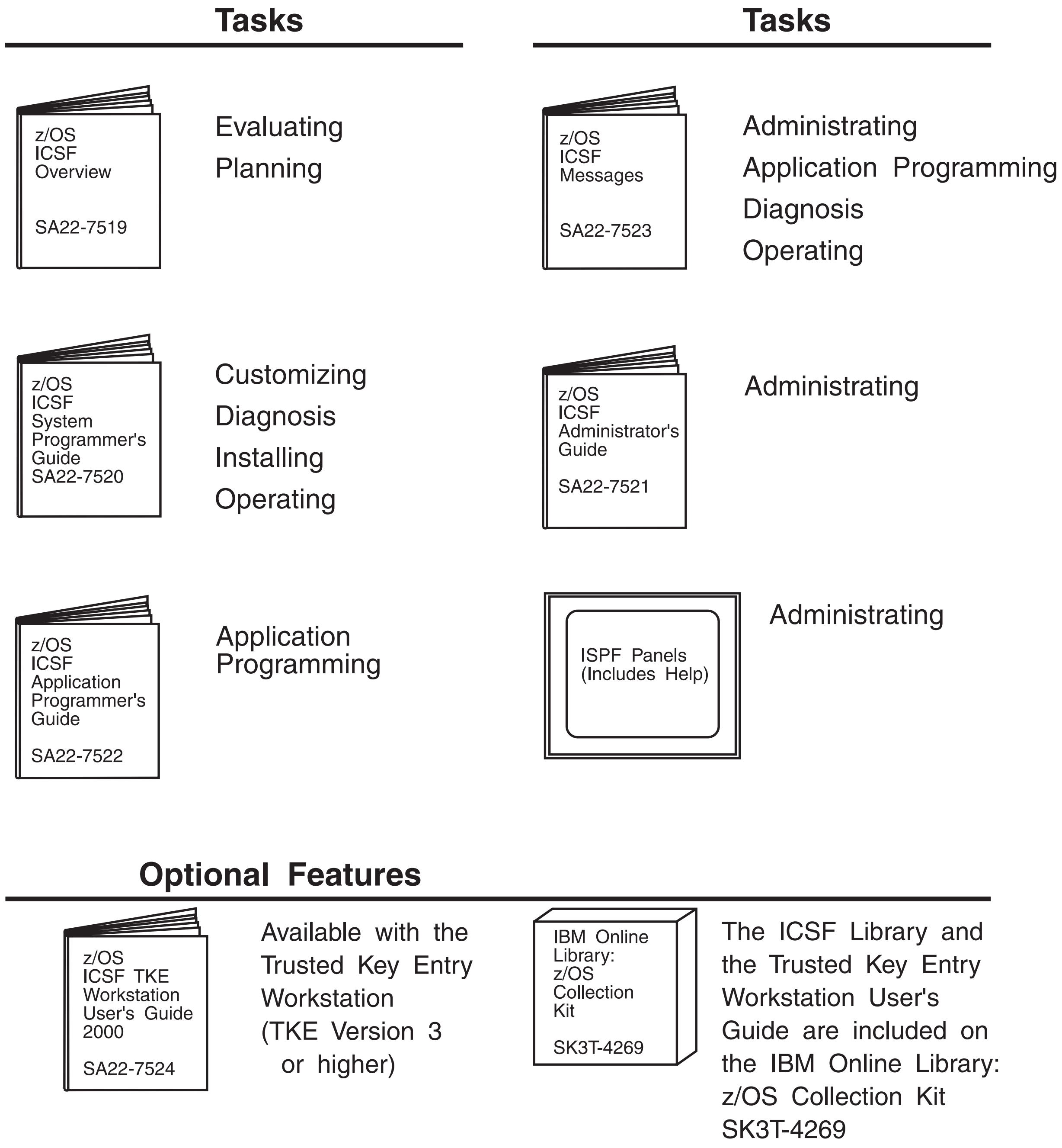

Tasks
z/OS ICSF Overview SA22-7519
Evaluating
Planning
z/OS ICSF System Programmer's Guide SA22-7520
Customizing
Diagnosis
Installing
Operating
z/OS ICSF Application Programmer's Guide SA22-7522
Application Programming
Tasks
z/OS ICSF Messages SA22-7523
Administrating
Application Programming
Diagnosis
Operating
z/OS ICSF Administrator's Guide SA22-7521
Administrating
ISPF Panels (Includes Help)
Administrating
Optional Features
z/OS ICSF TKE Workstation User's Guide 2000 SA22-7524
Available with the Trusted Key Entry Workstation (TKE Version 3 or higher)
IBM Online Library: z/OS Collection Kit SK3T-4269
The ICSF Library and the Trusted Key Entry Workstation User's Guide are included on the IBM Online Library: z/OS Collection Kit SK3T-4269

Figure 1. The z/OS ICSF Library

Using LookAt to look up message explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages, system abends, and some codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can access LookAt from the Internet at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html` or from anywhere in z/OS where you can access a TSO command line (for example, TSO prompt, ISPF, z/OS UNIX System Services running OMVS).

To find a message explanation on the Internet, go to the LookAt Web site and simply enter the message identifier (for example, `IAT1836` or `IAT*`). You can select a specific release to narrow your search. You can also download code from the *z/OS Collection*, SK3T-4269 and the LookAt Web site so you can access LookAt from a PalmPilot (Palm VIIx suggested).

To use LookAt as a TSO command, you must have LookAt installed on your host system. You can obtain the LookAt code for TSO from a disk on your *z/OS Collection*, SK3T-4269 or from the LookAt Web site. To obtain the code from the LookAt Web site, do the following:

1. Go to http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html.
2. Click the **News** button.
3. Scroll to **Download LookAt Code for TSO and VM.**
4. Click the ftp link, which will take you to a list of operating systems. Select the appropriate operating system. Then select the appropriate release.
5. Find the **lookat.me** file and follow its detailed instructions.

To find a message explanation from a TSO command line, simply enter: **lookat** *message-id*. LookAt will display the message explanation for the message requested.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in *z/OS MVS Routing and Descriptor Codes*. For such messages, LookAt prompts you to choose which book to open.

Accessing licensed books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at:

```
http://www.ibm.com/servers/resourcelink
```

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link Web userid and password, and a key code. With your z/OS order you received a memo that includes this key code.

To obtain your IBM Resource Link Web userid and password log on to:

```
http://www.ibm.com/servers/resourcelink
```

To register for access to the z/OS licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **User Profiles** located on the left-hand navigation bar.
3. Click on **Access Profile.**
4. Click on **Request Access to Licensed books.**
5. Supply your key code where requested and click on the **Submit** button.

If you supplied the correct key code you will receive confirmation that your request is being processed. After your request is processed you will receive an e-mail confirmation.

Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **Library**.
3. Click on **zSeries**.
4. Click on **Software**.
5. Click on **z/OS**.
6. Access the licensed book by selecting the appropriate element.

Do You Have Problems, Comments, or Suggestions?

Your suggestions and ideas can contribute to the quality and the usability of this book. If you have problems using this book, or if you have suggestions for improving it, complete and mail the Reader's Comment Form found at the back of the book.

Summary of changes

Summary of changes
for SA22-7522-02
z/OS Version 1 Release 3

This book contains information previously presented in *z/OS ICSF Application Programmer's Guide*, SA22-7522-01, which supports z/OS Version 1 Release 2.

New information

- Access Control Points
  - UKPT - PIN Verify, PIN Translate
- Callable services - The following new callable services perform encryption using the AES algorithm. AES encryption is only allowed if the CCC is enabled for triple DES. Only clear key support is provided.
  - Symmetric Key Decipher (CSNBSYD) - Deciphers data in an address space or a data space using the cipher block chaining or electronic code book modes.
  - Symmetric Key Encipher (CSNBSYE) - Enciphers data in an address space or a data space using the cipher block chaining or electronic code book modes.
- ICSF Setup
  - ICSF setup for E-Delivery delivery has been added. A sample ICSF options dataset, CSFPRM01, has been added to SYS1.SAMPLIB for the purpose of setting master keys by means of batch processing.
  - A sample CKDS allocation job (member CSFCKDS) has been added to SYS1.SAMPLIB.
  - A sample PKDS allocation job (member CSFPKDS) has been added to SYS1.SAMPLIB.
  - Samples for CSFSTART (ICSF Startup Procedures) has been added.
  - Sample JCL (CSFSETMK) for E-Delivery default passphrase has been added.
- Support to enable RMF to provide performance measurements on selected ICSF services and functions that use Direct Access Crypto (DAC) CCF instructions has been added.
- An appendix with z/OS product accessibility information has been added.

Changed information

- Callable services
  - Control Vector Generate (CSNBCVG) - *rule_array* enhanced to support the UKPT keyword.
  - Key Token Build (CSNBKTB) - *rule_array* enhanced to support the UKPT keyword.
  - Encrypted PIN Translate (CSNBPTR) - *rule_array* enhanced to support UKPT keywords UKPTIPIN, UKPTOPIN, and UKPTBOTH.
  - Encrypted PIN Verify (CSNBPVR) - *rule_array* enhanced to support UKPT keyword UKPTIPIN.
  - Symmetric Key Export (CSNDSYX) - a new *rule_array* keyword, PKCSOAEP, has been added. This keyword specifies the method found in RSA PKCS #1V2 OAEP.

  - Symmetric Key Generate (CSNDSYG) - a new *rule_array* keyword, PKCSOAEP, has been added. This keyword specifies the method found in RSA PKCS #1V2 OAEP.
  - Symmetric Key Import (CSNDSYI) - a new *rule_array* keyword, PKCSOAEP, has been added. This keyword specifies the method found in RSA PKCS #1V2 OAEP.
- The ICSF TSO panels have been updated to enhance usability:
  - Coprocessor management functions have been combined onto one panel
  - Master key management/CKDS functions combined onto one panel
  - TKE TSO utilities combined onto one panel
  - Primary panel simplified
  - New utility added to generate master key values from a pass phrase

This book contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Starting with z/OS V1R2, you may notice changes in the style and structure of some content in this book—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our books.

Summary of changes for SA22-7522-01 z/OS Version 1 Release 2

This book contains information previously presented in *z/OS ICSF Application Programmer's Guide*, SA22-7522-00, which supports z/OS Version 1 Release 1.

New information

- Callable services
  - PKA Key Token Change (CSNDKTC) callable service - This service changes PKA internal key tokens (RSA and DSS) from encipherment with the old PCI Cryptographic Coprocessor asymmetric-keys master key to encipherment with the current PCI Cryptographic Coprocessor asymmetric-keys master key.
  - Secure Messaging for Keys (CSNBSKY) callable service - This service encrypts a text block, including a clear key value decrypted from an internal or external DES token.
  - Secure Messaging for PINs (CSNBSPN) callable service - This service encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.
- Installation Options Data Set
  - PKDSCACHE, an installation option, defines the size of the PKDS Cache in records. The PKDS cache improves performance as it facilitates access to frequently used records. Specify *n* as a decimal value from 0 to 256. If *n* is zero, no cache will be implemented. If PKDSCACHE is not specified, the default value is 64. PKDSCACHE can be implemented on OS/390 V2 R10 and z/OS V1 R1 by installing APAR OW48568.
  - When specifying parameter values within parentheses, leading and trailing blanks are ignored. Embedded blanks may cause unpredictable results.

- PCI Cryptographic Accelerator (PCICA) support has been added. If a PCI Cryptographic Accelerator is available, clear RSA key processing in the CSFDPKD service will be routed to the PCI Cryptographic Accelerator. If you have a PCI Cryptographic Accelerator online, toleration APAR OW49402 is required on lower levels of ICSF (OS/390 V2 R9, OS/390 V2 R10 and z/OS V1 R1).
- Support to REENCIPHER PKDS and ACTIVATE PKDS has been added to the Master Key Management Panels. The new utility, CSFPUTIL, can also be used to reencipher the PKDS from the old asymmetric-keys master key to the current master key and to activate the reenciphered PKDS. Toleration APAR OW49386 is required on the following systems in order to activate the re-enciphered PKDS:
  - HCRP210 (standalone), HCRP220(OS/390 V2 R6, OS/390 V2 R7, OS/390 V2 R8), HCRP230 (OS/390 V2 R9), and HCR7703 (OS/390 V2 R10 and z/OS V1 R1)
- UDX support - Support for writing your own UDX has been added.

Changed information

- Beginning in z/OS V1 R2, the DOMAIN parameter is an optional parameter in the installation options data set. It is, however, required if more than one domain is specified as the usage domain on the PR/SM panels or if running in native mode. If specified in the options data set, it will be used and it must be one of the usage domains for the LPAR. If DOMAIN is not specified in the options data set, ICSF determines which domains are available in this LPAR. If only one domain is defined for the LPAR, ICSF will use it. If more than one is available, ICSF will issue error message ″CSFM409E MULTIPLE DOMAINS AVAILABLE. SELECT ONE IN THE OPTIONS DATA SET.″
- Callable services
  - MAXLEN parameter checking has been eliminated for the following services:
    - Encipher (CSNBENC and CSNBENC1)
    - Decipher (CSNBDEC and CSNBDEC1)
    - MAC generate (CSNBMGN and CSNBMGN1)
    - MAC verify (CSNBMVR and CSNBMVR1)
    - Ciphertext translate (CSNBCTT and CSNBCTT1)
    - MDC generate (CSNBMDG and CSNBMDG1)

    The MAXLEN parameter is also no longer enforced in the CUSP compatibility CIPHER service. The MAXLEN parameter may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647). If a value greater than this is specified, an error will result and ICSF will not start.
- Pass Phrase Initialization now allows uninitialized PCI Cryptographic Coprocessors to be initialized without processing all Cryptographic Coprocessors. A new panel option (Initialize new PCICC Only) has been added to the Pass Phrase Initialization panel to allow the initialization of the new PCI Cryptographic Coprocessors.

Deleted information

- Message IEC161I has been eliminated during the first time startup of ICSF.
- The following reason codes for ICSF/MVS X'18F' are being eliminated and will be replaced with operator messages.
  - Reason Code X'3C' - replaced by message CSFM105E
  - Reason Code X'48' - replaced by message CSFM120E

- Reason Code X'1B' - replaced by message CSFM410E
- Reason Code X'4B' - replaced by message CSFM107E
- Reason Code X'106' - If the CCC is all zeroes, abend X'18F' reason code 4A will occur. If the CCC does not exist, message CSFM113E will be displayed.

This book contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.

Part 1. IBM CCA Programming

This part of the book introduces programming for the IBM CCA, DES cryptography and PKA cryptography. It explains how to use DES and PKA callable services.

Chapter 1. Introducing Programming for the IBM CCA

ICSF provides access to cryptographic functions through callable services, which are also known as verbs. A callable service is a routine that receives control using a CALL statement in an application language.

Before invoking callable services in an application program, you must link them into the application program. See “Linking a Program with the ICSF Callable Services” on page 11.

To invoke the callable service, the application program must include a procedure call statement that has the entry point name and parameters for the callable service. The parameters that are associated with a callable service provide the only communication between the application program and ICSF.

Callable Service Syntax

This book uses a general call format to show the name of the ICSF callable service and its parameters. An example of that format is shown below:

```
CALL CSNBxxxx(return_code,
              reason_code,
              exit_data_length,
              exit_data,
              parameter_5,
              parameter_6,
              .
              .
              .
              parameter_N)
```

where CSNBxxxx is the name of the callable service. CSFXXX corresponds to CSNBxxx. (The ANSI services start with CSNAxxx and have corresponding CSFAxxx names. For the PKA services, which start with CSNDxxx and have corresponding CSFxxx names, see “Summary of the PKA Callable Services” on page 57.) The return code, reason code, exit data length, exit data, parameter 5 through parameter *N* represent the parameter list. The call generates a fixed length parameter list. You must supply the parameters in the order shown in the syntax diagrams. “Parameter Definitions” on page 6 describes the parameters in more detail.

ICSF callable services can be called from application programs written in a number of high-level languages as well as assembler. The high-level languages are:
- C
- COBOL
- FORTRAN
- PL/I

The ICSF callable services comply with the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface. The services can be invoked using the generic format, CSNBxxxx. Use the generic format if you want your application to work with more than one cryptographic product. Otherwise, use the **CSFxxxx** format.

Specific formats for the languages that can invoke ICSF callable services are as follows:

C

```
CSNBxxxx (return_code,reason_code,exit_data_length,exit_data,
parameter_5,...parameter_N)
```

COBOL

```
CALL 'CSNBxxxx' USING return_code,reason_code,exit_data_length,
exit_data,parameter_5,...parameter_N
```

FORTRAN

```
CALL CSNBxxxx (return_code,reason_code,exit_data_length,exit_data,
parameter_5,...parameter_N)
```

PL/I

```
DCL CSNBxxxx ENTRY OPTIONS(ASM);
CALL CSNBxxxx return_code,reason_code,exit_data_length,exit_data,
parameter_5,...parameter_N;
```

Assembler language programs must use standard linkage conventions when invoking ICSF callable services. An example of how an assembler language program can invoke a callable service is shown as follows:

```
CALL CSNBxxxx,(return_code,reason_code,exit_data_length,exit_data,
parameter_5,...parameter_N)
```

Coding examples using the high-level languages are shown in Appendix D, “Coding Examples”.

Callable Services with ALET Parameters

Some callable services have an alternate entry point (with ALET parameters—for data that resides in data spaces). They are in the format of *CSNBxxx1*:

Verb	Callable Service without ALET	Callable Service with ALET
Ciphertext translate	CSNBCTT	CSNBCTT1
Decipher	CSNBDEC	CSNBDEC1
Encipher	CSNBENC	CSNBENC1
MAC generate	CSNBMGN	CSNBMGN1
MAC verify	CSNBMVR	CSNBMVR1
MDC generate	CSNBMDG	CSNBMDG1
One way hash generate	CSNBOWH	CSNBOWH1

When choosing which service to use, consider the following:

- Callable services that do not have an ALET parameter require data to reside in the caller’s primary address space. A program using these services adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- Callable services that have an ALET parameter allow data to reside either in the caller’s primary address space or in a data space. This can allow you to encipher more data with one call. However, a program using these services does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

Rules for Defining Parameters and Attributes

The following rules apply to the callable services:

- Parameters are required and positional.
- Each parameter list has a fixed number of parameters.

- Each parameter is defined as an integer or a character string.
- Keywords passed to the callable services, such as TOKEN, CUSP, and FIRST can be in lower, upper, or mixed case. The callable services fold them to uppercase before using them.

Each callable service defines its own list of parameters. The entire list must be supplied on every call. If you do not use a specific parameter, you must supply that parameter with hexadecimal zeros or binary zeros.

Parameters are passed to the callable service. All information that is exchanged between the application program and the callable service is through parameters passed on the call.

Each parameter definition begins with the direction that the data flows and the attributes that the parameter must possess (called “type”). The following describes the direction.

Direction **Meaning**

Input The application sends (*supplies*) the parameter to the callable service. The callable service does not change the value of the parameter.

Output The callable service *returns* the parameter to the application program. The callable service may have changed the value of the parameter on return.

Input/Output The application sends (*supplies*) the parameter to the callable service. The callable service may have changed the value of the parameter on return.

The following describes the attributes or type.

Type **Meaning**

Integer (I) A 4-byte (32-bit), twos complement, binary number that has sign significance.

String A series of bytes where the sequence of the bytes must be maintained. Each byte can take on any bit configuration. The string consists only of data bytes. No string terminators, field-length values, or type-casting parameters are included. The maximum size of a string is X'7FFFFFFF' or 2 gigabytes. In some of the callable services, the length of some string data has an upper bound defined by the installation.

Alphanumeric character string
A string of bytes in which each byte represents characters from the following set:

Character	EBCDIC Value	Character	EBCDIC Value	Character	EBCDIC Value
A-Z		(	X'4D'	/	X'61'
a-z		)	X'5D'	,	X'6B'
0-9		+	X'4E'	%	X'6C'
Blank	X'40'	&	X'50'	?	X'6F'
*	X'5C'	.	X'4B'	:	X'7A'
<	X'4C'	;	X'5E'	=	X'7E'
>	X'6E'	-	X'60'	'	X'7D'

Parameter Definitions

This section describes the following parameters, which are used by most of the callable services:

- *Return_code*
- *Reason_code*
- *Exit_data_length*
- *Exit_data*
- *Key_identifier*

Note: The *return_code* parameter, the *reason_code* parameter, the *exit_data_length* parameter, and the *exit_data* parameter are used with every callable service.

Return and Reason Codes

Return_code and *reason_code* parameters return integer values upon completion of the call.

Return_code

The return code parameter contains the general results of processing as an integer.

Table 1 shows the standard return code values that the callable services return. A complete list of return codes is shown in Appendix A, “ICSF and TSS Return and Reason Codes”.

Table 1. Standard Return Code Values From ICSF Callable Services

Value Hex (Decimal)	Meaning
00 (00)	Successful. Normal return.
04 (04)	A warning. Execution was completed with a minor, unusual event encountered.
08 (08)	An application error occurred. The callable service was stopped due to an error in the parameters. Or, another condition was encountered that needs to be investigated.
0C (12)	Error. ICSF is not active or an environment error was detected.
10 (16)	System error. The callable service was stopped due to a processing error within the software or hardware.

Generally, CUSP and PCF macros will receive identical error return codes if they execute on CUSP/PCF or on ICSF. A single exception has been noted: if a key is installed on the ICSF CKDS with the correct label but with the wrong key type, CUSP/PCF issues a return code of 8, indicating that the key type was incorrect. ICSF issues a return code of 12, indicating that the key could not be found.

Reason_code

The reason code parameter contains the results of processing as an integer. You can specify which set of reason codes (ICSF or TSS) are returned from callable services. The default value is ICSF. For more information about the REASONCODES installation option, see *z/OS ICSF System Programmer’s Guide*. Different results are assigned to unique reason code values under a return code.

A list of reason codes is shown in Appendix A, “ICSF and TSS Return and Reason Codes”.

Exit Data Length and Exit Data

The following describes the *exit_data_length* and *exit_data* parameters. The parameters are input to all callable services. (Although all services require these parameters, several services ignore them. Installation exits are not enabled for the following callable services: code conversion, character/nibble conversion, X9.9 data editing, and some PKA callable services.

ICSF provides two installation exits for each callable service. The preprocessing exit is invoked when an application program calls a callable service, but before the callable service starts processing. For example, this exit is used to check or change parameters passed on the call or to stop the call. It can also be used to perform additional security checks.

The post-processing exit is invoked when the callable service has completed processing, but before the callable service returns control to the application program. For example, this exit can be used to check and change return codes from the callable service or perform clean-up processing.

For more information about the exits, see *z/OS ICSF System Programmer's Guide*.

Exit_data_length

The integer that has the string length of the data passed to the exit. The data is identified in the following *exit_data* parameter.

Exit_data

The installation exit data string that is passed to the callable service's preprocessing exit. The installation exit can use the data for its own processing.

Key Identifier for Key Token

A *key identifier* for a key token is an area that contains one of the following:

- **Key label** identifies keys that are in the CKDS or PKDS. Ask your ICSF administrator for the key labels that you can use.
- **Key token** can be either an internal key token, an external key token, or a null key token. Key tokens are generated by an application (for example, using the key generate callable service), or received from another system that can produce external key tokens.

  An **internal key token** can be used only on ICSF because the master key encrypts the key value. Internal key tokens contain keys in operational form only.

  An **external key token** can be exchanged with other systems because a transport key that is shared with the other system encrypts the key value. External key tokens contain keys in either exportable or importable form.

  A **null key token** can be used to import a key from a system that cannot produce external key tokens. A null key token contains a key encrypted under an importer key-encrypting key but does not contain the other information present in an external key token.

The term *key identifier* is used when a parameter could be one of the above items and to indicate that different inputs are possible. For example, you may want to specify a specific parameter as either an internal key token or a key label. The key label is, in effect, an indirect reference to a stored internal key token.

Key Label: If the first byte of the key identifier is greater than X'40', the field is considered to be holding a **key label**. The contents of a key label are interpreted as a pointer to a CKDS or PKDS key entry. The key label is an indirect reference to an internal key token.

A key label is specified on callable services with the *key_identifier* parameter as a 64-byte character string, left-justified, and padded on the right with blanks. In most cases, the callable service does not check the syntax of the key label beyond the first byte. One exception is the key record create callable service which enforces the KGUP rules for key labels unless syntax checking is bypassed by a preprocessing exit.

A key label has the following form:

Offset	**Length**	**Data**
00-63	64	Key label name

Invocation Requirements

Applications that use ICSF callable services must meet the following invocation requirements:

- Data can be located above or below 16Mb but must be 31-bit addressable
- Problem or supervisor state
- Any PSW key
- Task mode or Service Request Block (SRB) mode
- No mode restrictions
- Enabled for interrupts

Note: The dynamic CKDS update services have two additional restrictions.
- The caller must be in task mode, not SRB mode.
- The caller must not be in cross-memory mode.

Security Considerations

Your installation can use the Security Server (RACF) or an equivalent product to control who can use ICSF callable services or key labels. Before using an ICSF callable service or a key label, ask your security administrator to ensure that you have the necessary authorization.

Performance Considerations

In most cases, the z/OS operating system Dispatcher provides optimum performance. However, if your application makes extensive use of ICSF functions, you should consider using one or both of the following:

- If your application runs in SRB mode, you should consider scheduling an SRB to run on a processor with cryptographic feature installed (using the FEATURE=CRYPTO keyword on the SCHEDULE macro). For more information on the SCHEDULE macro, refer to *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.
- Use the IEAAFFN callable service (processor affinity) to avoid system overhead in selecting which processor your program (specifically, a particular TCB in the application) runs in. Note that you do **not** have to use the IEAAFFN service to ensure that the system runs a program on a processor with a cryptographic feature; the system ensures that automatically. However, you can avoid some of the system overhead involved in the selection process by using the IEAAFFN service, thus improving the program’s performance. For more information on using the IEAAFFN callable service, refer to *z/OS MVS Programming: Callable Services for HLL*.

IBM recommends that you run applications first without using these options. Consider these options when you are tuning your application for performance. Use these options only if they improve the performance of your application.

Special Secure Mode

Special secure mode is a special processing mode in which:

- The secure key import and multiple secure key import callable services, which works with clear keys, can be used.
- The Clear PIN Generate and Clear PIN Generate Alternate callable services, which works with clear PINs, can be used.
- The Symmetric Key Generate callable service with the ″IM″ keyword (the DES enciphered key is enciphered by an IMPORTER key) can be used.
- The key generator utility program (KGUP) can be used to enter clear keys into the CKDS.

To use special secure mode, several conditions must be met.

- The installation options data set must specify YES for the SSM installation option.

  For information about specifying installation options, see *z/OS ICSF System Programmer's Guide*.
- The environmental control mask (ECM) must be configured to permit special secure mode.

  The ECM is a 32-bit mask defined for each cryptographic domain during hardware installation. The second bit in this mask must have been turned on to enable special secure mode. The default is to have this bit turned on in the ECM. The bit can only be turned off/on through the optional TKE Workstation.
- If you are running in LPAR mode, special secure mode must be enabled.

  On S/390 Enterprise Servers, the S/390 Multiprise, and the IBM @server zSeries, you enable special secure mode during activation using the Crypto page of the Customize Activation Profiles task. After activation, you can enable or disable special secure mode on the Change LPAR Crypto task. Both of these tasks can be accessed from the Hardware Master Console.

For S/390 Enterprise Servers, the OS/390 Multiprise, and the @server zSeries with TKE, TKE can disable/enable special secure mode. For more information about entering clear keys, see *z/OS ICSF Administrator's Guide*.

Using the Callable Services

This section discusses how ICSF callable services use the different key types and key forms. It also provides suggestions on what to do if there is a problem.

ICSF provides callable services that perform cryptographic functions. You call and pass parameters to a callable service from an application program. Besides the callable services ICSF provides, you can write your own callable services called *installation-defined callable services*. **Note that only an experienced system programmer should attempt to write an installation-defined callable service**.

To write an installation-defined callable service, you must first write the callable service and link-edit it into a load module. Then define the service in the installation options data set.

You must also write a service stub. To execute an installation-defined callable service, you call a service stub from your application program. In the service stub, you specify the service number that identifies the callable service.

For more information about installation-defined callable services, see *z/OS ICSF System Programmer's Guide*.

When the Call Succeeds

If the return code is **0**, ICSF has successfully completed the call. If a reason code other than 0 is included, refer to Appendix A, "ICSF and TSS Return and Reason Codes" on page 335, for additional information. For instance, reason code 10000 indicates the key in the key identifier (or more than one key identifier, for services that use two internal key identifiers) has been reenciphered from encipherment under the old master key to encipherment under the current master key. Keys in external tokens are not affected by this processing because they contain keys enciphered under keys other than the host master key. If you manage your key identifiers on disk, then reason code 10000 should be a "trigger" to store these updated key identifiers back on disk.

Your program can now continue providing its function, but you may want to communicate the key that you used to another enterprise. This process is exporting a key.

If you want to communicate the key that you are using to a cryptographic partner, there are several methods to use:

- For DATA keys only, call the data key export callable service. You now have a DATA key type in exportable form.
- Call the key export callable service. You now have the key type in exportable form.
- When you use the key generate callable service to create your operational or importable key form, you can create an exportable form, **at the same time**, and you now have the key type, in exportable form, at the same time as you get the operational or importable form.

When the Call Does Not Succeed

Now you have planned your use of the ICSF callable services, made the call, but the service has completed with a return and reason codes other than zero.

If the return code is **4**, there was a minor problem. For example, reason code 8004 indicates the trial MAC that was supplied does not match the message text provided.

If the return code is **8**, there was a problem with one of your parameters. Check the meaning of the reason code value, correct the parameter, and call the service again. You may go through this process several times before you succeed.

If the return code is **12**, ICSF is not active, or has no access to cryptographic units, or has an environmental problem. Check with your ICSF administrator.

If the return code is **16**, the service has a serious problem that needs the help of your system programmer.

There are several reason codes that can occur **after** you have fully debugged and tested your program. For example:

- Reason code 10004 indicates that you provided a key identifier that holds a key enciphered under a host master key. The host master key is not installed in the cryptographic unit. If this happens, you have to go back and import your importable key form again and call the service again. You need to build this flow into your program logic.
- Reason code 10012 indicates a key corresponding to the label that you specified is not in the CKDS or PKDS. Check with your ICSF administrator to see if the label is correct.

Return and reason codes are described in Appendix A, “ICSF and TSS Return and Reason Codes” on page 335.

Linking a Program with the ICSF Callable Services

To link the ICSF callable services into an application program, you can use the following sample JCL statements.In the SYSLIB concatenation, include the CSF.SCSFMOD0 module in the link edit step.

```
//LKEDENC  JOB
//*-----------------------------------------------------------------*
//*                                                                 *
//*  The JCL links the ICSF encipher callable service, CSNBENC,   *
//*  into an application called ENCIPHER.                           *
//*                                                                 *
//*-----------------------------------------------------------------*
//LINK     EXEC PGM=IEWL,
//   PARM='XREF,LIST,LET'
//SYSUT1   DD   UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=CSF.SCSFMOD0,DISP=SHR        * SERVICES ARE IN HERE
//SYSLMOD  DD DSN=MYAPPL.LOAD,DISP=SHR         * MY APPLICATION LIBRARY
//SYSLIN   DD DSN=MYAPPL.ENCIPHER.OBJ,DISP=SHR * MY ENCIPHER PROGRAM
//         DD *
      ENTRY ENCIPHER
  NAME ENCIPHER(R)
/*
```

Chapter 2. Introducing DES Cryptography and Using DES Callable Services

The Integrated Cryptographic Service Facility protects data from unauthorized disclosure or modification. ICSF protects data stored within a system, stored in a file off a system on magnetic tape, and sent between systems. ICSF also authenticates the identity of customers in the financial industry and authenticates messages from originator to receiver. It uses cryptography to accomplish these functions.

ICSF provides access to cryptographic functions through callable services. A callable service is a routine that receives control using a CALL statement in an application language. Each callable service performs one or more cryptographic functions, including:

- Generating and managing cryptographic keys
- Enciphering and deciphering data with encrypted keys using either the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES), or the Commercial Data Masking Facility (CDMF)
- Transforming a CDMF DATA key to a transformed shortened DES key
- Reenciphering text from encryption under one key to encryption under another key
- Encoding and decoding data with clear keys
- Generating random numbers
- Ensuring data integrity and verifying message authentication
- Generating, verifying, and translating personal identification numbers (PINs) that identify a customer on a financial system

This chapter provides an overview of the DES cryptographic functions provided in ICSF, explains the functions of the cryptographic keys, and introduces the topic of building key tokens.

Functions of the DES Cryptographic Keys

ICSF provides functions to create, import, and export DES keys. This section gives an overview of these cryptographic keys. Detailed information about how ICSF organizes and protects keys is in *z/OS ICSF Administrator's Guide*.

Key Separation

The cryptographic feature controls the use of keys by separating them into unique types, allowing you to use a specific type of key only for its intended purpose. For example, a key used to protect data cannot be used to protect a key.

An ICSF system has only one DES master key. However, to provide for key separation, the cryptographic feature automatically encrypts each type of key under a unique variation of the master key. Each variation of the master key encrypts a different type of key. Although you enter only one master key, you have a unique master key to encrypt all other keys of a certain type.

Note: In CUSP/PCF, key separation applies only to keys enciphered under the master key (keys in operational form). In ICSF, key separation also applies to keys enciphered under transport keys (keys in importable or exportable form). This allows the creator of a key to transmit the key to another system and to enforce its use at the other system.

Master Key Variant

Whenever the master key is used to encipher a key, the cryptographic coprocessor produces a variation of the master key according to the type of key the master key will encipher. These variations are called *master key variants*. The cryptographic coprocessor creates a master key variant by exclusive ORing a fixed pattern, called a *control vector*, onto the master key. A unique control vector is associated with each type of key. For example, all the different types of data-encrypting, PIN, MAC, and transport keys are each exclusive ORed with a unique control vector. The different key types are described in "Types of Keys" on page 17.

Each master key variant protects a different type of key. It is similar to having a unique master key protect all the keys of a certain type.

The master key, in the form of master key variants, protects keys operating on the system. A key can be used in a cryptographic function only when it is enciphered under a master key. When systems want to share keys, transport keys are used to protect keys sent outside of systems. When a key is enciphered under a transport key, the key cannot be used in a cryptographic function. It must first be brought on to a system and enciphered under the system's master key, or exported to another system where it will then be enciphered under that system's master key.

Transport Key Variant

Like the master key, ICSF creates variations of a transport key to encrypt a key according to its type. This allows for key separation when a key is transported off the system. A *transport key variant*, also called *key-encrypting key variant*, is created the same way a master key variant is created. The transport key's clear value is exclusive ORed with a control vector associated with the key type of the key it protects.

Note: To exchange keys with systems that do not recognize transport key variants, ICSF allows you to encrypt selected keys under a transport key itself, not under the transport key variant.

Key Forms

A key that is protected under the master key is in *operational form*, which means ICSF can use it in cryptographic functions on the system.

When you store a key with a file or send it to another system, the key is enciphered under a transport key rather than the master key because, for security reasons, the key should no longer be active on the system. When ICSF enciphers a key under a transport key, the key is not in operational form and cannot be used to perform cryptographic functions.

When a key is enciphered under a transport key, the sending system considers the key in *exportable form*. The receiving system considers the key in *importable form*. When a key is reenciphered from under a transport key to under a system's master key, it is in operational form again.

Enciphered keys appear in three forms. The form you need depends on how and when you use a key.

- **Operational** key form is used at the local system. Many callable services can *use* an operational key form.

  The key token build, key generate, key import, data key import, clear key import, multiple clear key import, secure key import, and multiple secure key import callable services can *create* an operational key form.
- **Exportable** key form is transported to another cryptographic system. It can only be passed to another system. The ICSF callable services cannot use it for cryptographic functions. The key generate, data key export, and key export callable services produce the exportable key form.
- **Importable** key form can be transformed into operational form on the local system. The key import callable service (CSNBKIM) and the Data Key Import callable service (CSNBDKM) can *use* an importable key form. Only the key generate callable service (CSNBKGN) can *create* an importable key form. The secure key import (CSNBSKI) and multiple secure key import (CSNBSKM) callable services can convert a clear key into an importable key form.

For more information about the key types, see either “Functions of the DES Cryptographic Keys” on page 13 or the *z/OS ICSF Administrator’s Guide*. See “Key Forms and Types Used in the Key Generate Callable Service” on page 35 for more information about key form.

DES Key Flow

The conversion from one key to another key is considered to be a one-way flow. An operational key form cannot be turned back into an importable key form. An exportable key form cannot be turned back into an operational or importable key form. The flow of ICSF key forms can only be in one direction:

```
IMPORTABLE —to→ OPERATIONAL —to→ EXPORTABLE
```

Key Token

A key token is a 64-byte field composed of a key value and control information. The control information is assigned to the key when ICSF creates the key. The key token can be either an internal key token, an external key token, or a null key token. Through the use of key tokens, ICSF can do the following:
- Support continuous operation across a master key change
- Control use of keys in cryptographic services

If the first byte of the key identifier is X'01', the key identifier is interpreted as an **internal key token**. An internal key token is a token that can be used only on the ICSF system that created it (or another ICSF system with the same host master key). It contains a key that is encrypted under the master key.

An application obtains an internal key token by using one of the callable services such as those listed below. The callable services are described in detail in Chapter 4, “Managing DES Cryptographic Keys”.
- Key generate
- Key import
- Secure key import
- Multiple secure key import
- Clear key import
- Multiple clear key import
- Key record read
- Key token build
- Data Key Import

The master key may be dynamically changed between the time that you invoke a service, such as the key import callable service to obtain a key token, and the time that you pass the key token to the encipher callable service. When a change to the master key occurs, ICSF reenciphers the caller's key from under the old master key to under the new master key. A Return Code of 0 with a reason code of 10000 notifies you that ICSF reenciphered the key. For information on reenciphering the CKDS or the PKDS, see *z/OS ICSF Administrator's Guide*.

Attention: If an internal key token held in user storage is not used while the master key is changed twice, the internal key token is no longer usable. (See "Other Considerations" on page 18 for additional information.)

For debugging information, see Appendix B, "Key Token Formats" for the format of an internal key token.

If the first byte of the key identifier is X'02', the key identifier is interpreted as an **external key token**. By using the external key token, you can exchange keys between systems. It contains a key that is encrypted under a key-encrypting key.

An external key token contains an encrypted key and control information to allow compatible cryptographic systems to:

- Have a standard method of exchanging keys
- Control the use of keys through the control vector
- Merge the key with other information needed to use the key

An application obtains the external key token by using one of the callable services such as those listed below. They are described in detail in Chapter 4, "Managing DES Cryptographic Keys".

- Key generate
- Key export
- Data key export

For debugging information, see Appendix B, "Key Token Formats" for the format of an external key token.

If the first byte of the key identifier is X'00', the key identifier is interpreted as a **null key token**. Use the null key token to import a key from a system that cannot produce external key tokens. That is, if you have an 8- to 16-byte key that has been encrypted under an importer key, but is not imbedded within a token, place the encrypted key in a null key token and then invoke the key import callable service to get the key in operational form.

For debugging information, see Appendix B, "Key Token Formats" for the format of a null key token.

Control Vector

A unique control vector exists for each type of key the master key enciphers. The cryptographic feature exclusive ORs the master key with the control vector associated with the type of key the master key will encipher. The control vector ensures that an operational key is only used in cryptographic functions for which it is intended. For example, the control vector for an input PIN-encrypting key ensures that such a key can be used only in the Encrypted PIN translate and Encrypted PIN verify functions.

Types of Keys

The cryptographic keys are grouped into the following categories based on the functions they perform.

- **DES master key.** The DES master key is a double-length (128 bits) key used only to encrypt other keys. The ICSF administrator installs and changes the DES master key (see *z/OS ICSF Administrator's Guide* for details). On S/390 Enterprise Servers and S/390 Multiprise and the IBM @server zSeries, the administrator does this by using the Clear Master Key Entry panels or the optional Trusted Key Entry (TKE) workstation.

  The master key always remains in a secure area in the cryptographic facility.

  It is used only to encipher and decipher keys. Other keys also encipher and decipher keys and are mostly used to protect cryptographic keys you transmit on external links. These keys, while on the system, are also encrypted under the master key.
- **SYM-MK master key.** The SYM-MK master key is a double-length (128-bit) key that is used only to encrypt other DES keys on the PCI Cryptographic Coprocessor. The ICSF administrator installs and changes the SYM-MK master key using either the ICSF panels or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the PCI Cryptographic Coprocessor. As with the DES master key, the SYM-MK master key is used only to encipher and decipher keys that are in operational form.
- **Data-encrypting keys.** The data-encrypting keys are single-length (64-bit), double-length (128-bit), or triple-length (192-bit) keys that protect data privacy. Single-length data-encrypting keys can also be used to encode and decode data and authenticate data sent in messages. If you intend to use a data-encrypting key for an extended period, you can store it in the CKDS so that it will be reenciphered if the master key is changed.

  You can use single-length data-encrypting keys in the encipher, decipher, encode, and decode callable services to manage data and also in the MAC generation and MAC verification callable services. Double-length and triple-length data-encrypting keys can be used in the encipher and decipher callable services for more secure data privacy.

  Single-length data-encrypting keys can be exported and imported using the ANSI X9.17 key management callable services.
- **Data-translation keys.** The data-translation keys are single-length (64 bits) keys used for the ciphertext translate callable service as either the input or the output data-transport key.
- **MAC keys.** The MAC keys are single-length (64 bits) and double-length (128 bits) keys used for the MAC generation and MAC verification callable services.
- **PIN keys.** The personal identification number (PIN) is a basis for verifying the identity of a customer across financial industry networks. PIN keys are used in cryptographic functions to generate, translate, and verify PINs, and protect PIN blocks. They are all double-length (128 bits) keys. PIN keys are used in the Clear PIN generate, Encrypted PIN verify, and Encrypted PIN translate callable services.

  For installations that do not support double-length 128-bit keys, effective single-length keys are provided. For a single-length key, the left key half of the key equals the right key half.

  "Managing Personal Authentication" on page 31 gives an overview of the PIN algorithms you need to know to write your own application programs.

- **Transport keys (or key-encrypting keys).** Transport keys are also known as key-encrypting keys. They are double-length (128 bits) keys used to protect keys when you distribute them from one system to another.

  There are four types of transport keys:

  - *Exporter or OKEYXLAT key-encrypting key* protects keys of any type that are sent from your system to another system. The exporter key at the originator is the same key as the importer key of the receiver.
  - *Importer or IKEYXLAT key-encrypting key* protects keys of any type that are sent from another system to your system. It also protects keys that you store externally in a file that you can import to your system later. The importer key at the receiver is the same key as the exporter key at the originator.

  Note: Transport keys replace local, remote, and cross keys used by CUSP/PCF.

  You use key-encrypting keys to protect keys that are transported using any of the following services: data key export, key export, key import, clear key import, multiple clear key import, secure key import, multiple secure key import, key generate, and key translate.

  For installations that do not support double-length key-encrypting keys, effective single-length keys are provided. For an effective single-length key, the clear key value of the left key half equals the clear key value of the right key half.
- **ANSI X9.17 key-encrypting keys.** These bidirectional key-encrypting keys are used exclusively in ANSI X9.17 key management. They are either single-length (64 bits) or double-length (128 bits) keys used to protect keys when you distribute them from one system to another according to the ANSI X9.17 protocol.
- **Key-Generating Keys.** Key-generating keys are double-length keys used to derive unique-key-per-transaction keys.

Other Considerations

The following are considerations for keys held in the cryptographic key data set (CKDS) or by applications.

- ICSF ensures that keys held in the CKDS are reenciphered during the master key change. Keys with a long life span (more than one master key change) should be stored in the CKDS.
- Keys enciphered under the host DES master key and held by applications are automatically reenciphered under a new master key as they are used. Keys with a short life span (for example, VTAM SLE data keys) do not need to be stored in the CKDS. However, if you have keys with a long life span and you do not store them in the CKDS, they should be enciphered under the importer key-encrypting key. The importer key-encrypting key itself should be stored in the CKDS.

Table 2 describes the key types. You can build, generate, import, or export key types DECIPHER, ENCIPHER, CIPHER, CVARDEC, and CVARPINE, but they will not be usable by ICSF in other services since they are not supported by the Cryptographic Coprocessor Feature.

Table 2. Descriptions of Key Types

Key Type	Meaning
AKEK	Single-length or double-length, bidirectional key-encrypting key used for the ANSI X9.17 key management callable services.

Table 2. Descriptions of Key Types (continued)

Key Type	Meaning
CIPHER	Used only to encrypt or decrypt data. Possible key lengths are single-length or double-length. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
CVARDEC	The TSS Cryptographic variable decipher verb uses a CVARDEC key to decrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.
CVARENC	Cryptographic variable encipher service uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.
CVARPINE	Used to encrypt a PIN value for decryption in a PIN-printing application. Possible key lengths are single-length or double-length.
CVARXCVL	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
CVARXCVR	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
DATA	Data encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAC	Used to specify a DATA-class key that will perform in the Encipher and Decipher callable services, but not in the MAC Generate or MAC Verify callable services. This is a double-length key.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
DECIPHER	Used only to decrypt data. Possible key lengths are single-length or double-length. DECIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
DKYGENKY	Used to generate a diversified key based on the key-generating key. This is a double-length key.
ENCIPHER	Used only to encrypt data. Possible key lengths are single-length or double-length. ENCIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
EXPORTER	Exporter key-encrypting key. Use this double-length key (including a DATA key) from the operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.

Table 2. Descriptions of Key Types (continued)

Key Type	Meaning
IMP-PKA	Double-length limited-authority importer key used to encrypt PKA private key values in PKA external tokens.
IPINENC	Double-length input PIN-encrypting key. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate, Encrypted PIN verify, and Clear PIN Generate Alternate services. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an IPINENC key.
KEYGENKY	Used to generate a key based on the key-generating key. This is a double-length key.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Output PIN-encrypting key. Use this double-length output key to translate PINs. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.
SECMSG	Used to encrypt PINs or keys in a secure message. This is a double-length key.

Clear Keys

A clear key is the base value of a key, and is not encrypted under another key. Encrypted keys are keys whose base value has been encrypted under another key.

There are four callable services you can use to convert a clear key to an encrypted key:

- To convert a clear key to an encrypted *data* key in operational form, use either the clear key import callable service or the multiple clear key import callable service.
- To convert a clear key to an encrypted key of any type, in operational or importable form, use either the secure key import callable service or the multiple secure key import callable service.

Note: The secure key import and multiple secure key import callable services can only execute in special secure mode.

Generating and Managing DES Keys

Using ICSF, you can generate keys using either the *key generator utility program* or the *key generate callable service*. The dynamic CKDS update callable services allow applications to directly manipulate the CKDS. ICSF provides callable services that support DES key management as defined by the IBM Common Cryptographic Architecture (CCA) and by the ANSI X9.17 standard. CDMF also supports such DES key management.

The next few sections describe the key generating and management options ICSF provides.

Key Generator Utility Program

The key generator utility program generates data, data-translation, MAC, PIN, and key-encrypting keys, and enciphers each type of key under a specific master key variant. After the KGUP generates a key, it stores it in the cryptographic key data set (CKDS).

Note: If you specify CLEAR, KGUP uses the random number generate and secure key import callable services rather than the key generate service.

You can access KGUP using ICSF panels. The KGUP path of these panels helps you create the JCL control statements to control the key generator utility program. When you want to generate a key, you can enter the ADD control statement and information, such as the key type on the panels. For a detailed description of the key generator utility program and how to use it to generate keys, see *z/OS ICSF Administrator's Guide*.

Common Cryptographic Architecture DES Key Management Services

ICSF provides callable services that support CCA key management for DES keys.

Clear Key Import Callable Service

This service imports a clear DATA key that is used to encipher or decipher data. It accepts a clear key and enciphers the key under the host master key, returning an encrypted DATA key in operational form in an internal key token.

Control Vector Generate Callable Service

The control vector generate callable service builds a control vector from keywords specified by the *key_type* and *rule_array* parameters.

Control Vector Translate Callable Service

The control vector translate callable service changes the control vector used to encipher an external key. Use of this service requires the optional PCI Cryptographic Coprocessor.

Cryptographic Variable Encipher Callable Service

The cryptographic variable encipher callable service uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. You can use this service to prepare a mask array for the control vector translate service. The plaintext must be a multiple of eight bytes in length. Use of this service requires the optional PCI Cryptographic Coprocessor.

Data Key Export Callable Service

This service reenciphers a DATA key from encryption under the master key to encryption under an exporter key-encrypting key, making it suitable for export to another system.

Data Key Import Callable Service

This service imports an encrypted source DES single-length or double-length DATA key and creates or updates a target internal key token with the master key enciphered source key. Use of this service requires the optional PCI Cryptographic Coprocessor.

Diversified Key Generate Callable Service

The diversified key generate service generates a key based on the key-generating key, the processing method, and the parameter supplied. The control vector of the key-generating key also determines the type of target key that can be generated. Use of this service requires the PCI Cryptographic Coprocessor.

Key Export Callable Service

This service reenciphers any type of key (except an AKEK or IMP-PKA key) from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key, making it suitable for export to another system.

Key Generate Callable Service

The key generate callable service generates data, data-translation, MAC, PIN, and key-encrypting keys. It generates a single key or a pair of keys. Unlike the key generator utility program, the key generate service does not store the keys in the CKDS where they can be saved and maintained. The key generate callable service returns the key to the application program that called it. The application program can then use a dynamic CKDS update service to store the key in the CKDS.

When you call the key generate callable service, include parameters specifying information about the key you want generated. Because the form of the key restricts its use, you need to choose the form you want the generated key to have. You can use the *key_form* parameter to specify the form. The possible forms are:

- **Operational,** if the key is used for cryptographic operations on the local system. Operational keys are protected by master key variants and can be stored in the CKDS or held by applications in internal key tokens.
- **Importable,** if the key is stored with a file or sent to another system. Importable keys are protected by importer key-encrypting keys.
- **Exportable,** if the key is transported or exported to another system and imported there for use. Exportable keys are protected by exporter key-encrypting keys and cannot be used by ICSF callable service.

Importable and exportable keys are contained in external key tokens. For more information on key tokens, refer to “Key Token” on page 15.

Key Import Callable Service

This service reenciphers a key (except an AKEK) from encryption under an importer key-encrypting key to encryption under the master key. The reenciphered key is in the operational form.

Key Part Import Callable Service

This service combines clear key of any key type and returns the combined key value either in an internal token or as an update to the CKDS. The PCI Cryptographic Coprocessor is required for all keys types except AKEK.

Key Test Callable Service

This service generates or verifies a secure cryptographic verification pattern for keys. A parameter indicates the action you want to perform.

The key to test can be in the clear or encrypted under a master key. The key test extended callable service works on keys encrypted under a KEK.

For generating a verification pattern, the service creates and returns a random number with the verification pattern. For verifying a pattern, you supply the random number from the call to the service that generated the pattern.

Key Token Build Callable Service

The key token build callable service is a utility you can use to create skeleton key tokens for AKEKs as input to the key generate or key part import callable service. You can also use this service to build CCA key tokens for all key types ICSF supports or to update the data encryption standard bits in a supplied DATA, IMPORTER, or EXPORTER token.

Key Translate Callable Service

This service uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment. Use of this service requires the optional PCI Cryptographic Coprocessor.

Multiple Clear Key Import Callable Service

This service imports a single-length, double-length, or triple-length clear DATA key that is used to encipher or decipher data. It accepts a clear key and enciphers the key under the host master key, returning an encrypted DATA key in operational form in an internal key token.

Multiple Secure Key Import Callable Service

This service enciphers a single-length, double-length, or triple-length clear key under the host master key or under an importer key-encrypting key. The clear key can then be imported as any of the possible key types. Triple-length keys can only be imported as DATA keys. This service can be used only when ICSF is in special secure mode and does not allow the import of an AKEK.

Prohibit Export Callable Service

This service modifies an operational key so that it cannot be exported. This callable service does not support NOCV key-encrypting keys, DATA, MAC, or MACVER keys with standard control vectors (for example, control vectors supported by the Cryptographic Coprocessor Feature). Use of this service requires the optional PCI Cryptographic Coprocessor.

Prohibit Export Extended Callable Service

This service updates the control vector in the external token of a key in exportable form so that the receiver node can import the key but not export it. When the key import callable service imports such a token, it marks the token as non-exportable. The key export callable service does not allow export of this token.

Random Number Generate Callable Service

The random number generate callable service creates a random number value to use in generating a key. The callable service uses cryptographic hardware to generate a random number for use in encryption.

Secure Key Import Callable Service

This service enciphers a clear key under the host master key or under an importer key-encrypting key. The clear key can then be imported as any of the possible key types. This service can be used only when ICSF is in special secure mode and does not allow the import of an AKEK.

Note: The PKA encrypt, PKA decrypt, symmetric key generate, symmetric key import, and symmetric key export callable services provide a way of distributing DES DATA keys protected under a PKA key. See Chapter 3, “Introducing PKA Cryptography and Using PKA Callable Services” on page 47 for additional information.

Symmetric Key Export Callable Service

This service transfers an application-supplied symmetric key (a DATA key) from encryption under the DES host master key to encryption under an application-supplied RSA public key. (There are two types of PKA public key tokens: RSA and DSS. This callable service can use only the RSA type.) The application-supplied DATA key must be an ICSF DES internal key token or the label of such a token in the CKDS. The symmetric key import callable service can import the PKA-encrypted form at the receiving node.

Symmetric Key Generate Callable Service

This service generates a symmetric key (that is, a DATA key) and returns it encrypted using DES and encrypted under an RSA public key token. (There are two types of PKA public key tokens: RSA and DSS. This callable service can use only the RSA type.)

The DES-encrypted key can be an internal token encrypted under a host DES master key, or an external form encrypted under a KEK. (You can use the symmetric key import callable service to import the PKA-encrypted form.)

Symmetric Key Import Callable Service

This service imports a symmetric (DES) DATA key enciphered under an RSA public key. (There are two types of PKA private key tokens: RSA and DSS. This callable service can use only the RSA type.) This service returns the key in operational form, enciphered under the DES master key.

Transform CDMF Key Callable Service

This service is available for S/390 Enterprise Servers and S/390 Multiprise only. It changes a CDMF DATA key in an internal or external token to a transformed shortened DES key. It ignores the input internal DES token markings and marks the output internal token for use in the DES. You need to use this service only if you have a CDMF or DES-CDMF system that needs to send CDMF-encrypted data to a DES-only system. The CDMF or DES-CDMF system must generate the key, shorten it, and pass it to the DES-only system.

If the input DATA key is in an external token, the operational KEK must be marked as DES or SYS-ENC. The service fails for an external DATA key encrypted under a KEK that is marked as CDMF.

User Derived Key Callable Service

This service generates a single-length or double-length MAC key, or updates an existing user-derived key. A single-length MAC key can be used to compute a Message Authentication Code (MAC) following the ANSI X9.9, ANSI X9.19, or the Europay, MasterCard, Visa (EMV) Specification MAC processing rules. A double-length MAC key can be used to compute a MAC following the ANSI X9.19 optional double MAC processing rule or the EMV rules.

Callable Services for Dynamic CKDS Update

ICSF provides the dynamic CKDS update services that allow applications to directly manipulate both the DASD copy and in-storage copy of the current CKDS.

Note: Applications using the dynamic CKDS update callable services can run concurrently with other operations that affect the CKDS, such as KGUP, CKDS conversion, REFRESH, and dynamic master key change. An operation can fail if it needs exclusive or shared access to the same DASD copy of the CKDS that is held shared or exclusive by another operation. ICSF provides serialization to prevent data loss from attempts at concurrent access, but your installation is responsible for the effective management of concurrent use of competing operations. Consult your system administrator or system programmer for your installation guidelines.

The syntax of the key record create, key record read, and key record write services is identical with the same services provided by the Transaction Security System security application programming interface. Key management applications that use these common interface verbs can run on both systems without change.

Key Record Create Callable Service

This service accepts a key label and creates a null key record in both the DASD copy and in-storage copy of the CKDS. The record contains a key token set to binary zeros and is identified by the key label passed in the call statement. The key label must be unique. Callers must be in task mode and cannot be in cross memory mode.

Before you can update a key record using either the dynamic CKDS update services or KGUP, that record must already exist in the CKDS. You can use either the key record create service, KGUP, or your key entry hardware to create the initial record in the CKDS.

Key Record Delete Callable Service

This service accepts a unique key label and deletes the associated key record from both the in-storage and DASD copies of the CKDS. This service deletes the entire record, including the key label from the CKDS. Callers must be in task mode and cannot be in cross memory mode to execute this service.

Key Record Read Callable Service

This service copies an internal key token from the in-storage CKDS to the application storage, where it may be used directly in other cryptographic services. Key labels specified with this service must be unique.

Key Record Write Callable Service

This service accepts an internal key token and a label and writes the key token to the CKDS record identified by the key label. The key label must be unique. Application calls to this service write the key token to both the DASD copy and in-storage copy of the CKDS, so the record must already exist in both copies of the CKDS. Callers must be in task mode and cannot be in cross memory mode.

Callable Services that Support Secure Sockets Layer (SSL)

The Secure Sockets Layer (SSL) protocol, developed by Netscape Development Corporation, provides communications privacy over the Internet. Client/server applications can use the SSL protocol to provide secure communications and prevent eavesdropping, tampering, or message forgery.

ICSF provides callable services that support the RSA-encryption and RSA-decryption of PKCS 1.2-formatted symmetric key data to produce symmetric session keys. These session keys can then be used to establish an SSL session between the sender and receiver.

PKA Decrypt Callable Service

The PKA decrypt callable service uses the corresponding private RSA key to unwrap the RSA-encrypted key and deformat the key value. This service then returns the clear key value to the application.

PKA Encrypt Callable Service

The PKA encrypt callable service encrypts a supplied clear key value under an RSA public key. Currently, the supplied key can be formatted using the PKCS 1.2 or ZERO-PAD methods prior to encryption.

System Encryption Algorithm

ICSF uses either the DES algorithm or the Commercial Data Masking Facility (CDMF) to encipher and decipher data. The CDMF defines a scrambling technique for data confidentiality. It is intended to be a substitute for DES for those customers who have been previously prohibited from receiving IBM products that support DES data confidentiality services. The CDMF data confidentiality algorithm is composed of two processes: a key shortening process and a standard DES process to encipher and decipher data.

Your system can be one of the following:
- DES
- CDMF
- DES-CDMF

A DES system protects data using a single-length, double-length, or triple-length DES data-encrypting key and the DES algorithm.

A CDMF system protects data using a single-length DES data-encrypting key and the CDMF. You input a standard single-length data-encrypting key to the encipher (CSNBENC) and decipher (CSNBDEC) callable services. The single-length data-encrypting key that is intended to be passed to the CDMF is called a CDMF key. Cryptographically, it is indistinguishable from a DES data-encrypting key. Before the key is used to encipher or decipher data, however, the Cryptographic Coprocessor Feature hardware cryptographically shortens the key as part of the CDMF process. This transformed, shortened data-encrypting key can be used only in the DES. (It must never be used in the CDMF; this would result in a double shortening of the key.) When used with the DES, a transformed, shortened data-encrypting key produces results identical to those that the CDMF would produce using the original single-length key.

A DES-CDMF system protects data using either the DES or the CDMF. The default is DES.

ICSF provides functions to mark internal IMPORTER, EXPORTER, and DATA key tokens with **data encryption algorithm bits.** IMPORTER and EXPORTER KEKs are marked when they are installed in operational form in ICSF. Your cryptographic key administrator does this. (See *z/OS ICSF Administrator's Guide* for details.) Whenever a DATA key is imported or generated in concert with a marked KEK, this marking is transferred to the DATA key token, unless the token copying function of the callable service is used to override the KEK marking with the marking of the key token passed. These data encryption algorithm bits internally drive the DES or CDMF for the ICSF encryption services. External key tokens are not marked with these data encryption algorithm bits.

IMPORTER and EXPORTER KEKs can have data encryption algorithm bit markings of CDMF (X'80'), DES (X'40'), or SYS-ENC (X'00'). DATA keys generated or imported with marked KEKs will also be marked. A CDMF-marked KEK will transfer a data encryption algorithm bit marking of CDMF (X'80') to the DATA key token. A DES-marked KEK will transfer a data encryption algorithm bit marking of DES (X'00') to the DATA key token. A SYS-ENC-marked KEK will transfer a CDMF

(X'80') marking to the DATA key token on a CDMF system, and a DES (X'00') marking to the DATA key token on DES-CDMF and DES systems.

Notes:

1. For the multiple secure key import callable service the token markings on the KEK are ignored. In this case, the algorithm choice specified in the rule array determines the markings on the DATA key.
2. Propagation of data encryption algorithm bits and token copying are only performed when the ICSF callable service is performed on the Cryptographic Coprocessor Feature. The PCI Cryptographic Coprocessor does not perform these functions.

Table 3 summarizes the data encryption algorithm bits by key type, and the algorithm they drive in the ICSF encryption services.

Table 3. Summary of Data Encryption Standard Bits

Algorithm	Key Type	Bits
CDMF	DATA	X'80'
	KEK	X'80'
DES	DATA	X'00'
	KEK	X'40'
System Default Algorithm	KEK	X'00'

For CUSP/PCF users, your system programmer specifies a default encryption mode of DES or CDMF when installing ICSF. (See *z/OS ICSF System Programmer's Guide* for details.)

ANSI X9.17 Key Management Services

The ANSI X9.17 key management standard defines a process for protecting and exchanging DES keys. The ANSI X9.17 standard defines methods for generating, exchanging, using, storing, and destroying these keys. ANSI X9.17 keys are protected by the processes of *notarization* and *offsetting*, instead of control vectors. In addition to providing services to support these processes, ICSF also defines and uses an optional process of *partial notarization*.

Offsetting involves exclusive-ORing a key-encrypting key with a counter. The counter, a 56-bit binary number that is associated with a key-encrypting key and contained in certain ANSI X9.17 messages, prevents either a replay or an out-of-sequence transmission of a message. When the associated AKEK is first used, the application initializes the counter. With each additional use, the application increments the counter.

Notarization associates the identities of a pair of communicating parties with a cryptographic key. The notarization process cryptographically combines a key with two 16-byte quantities, the origin identifier and the destination identifier, to produce a notarized key. The notarization process is completed by offsetting the AKEK with a counter.

ICSF makes it possible to divide the AKEK notarization process into two steps. In the first step, partial notarization, the AKEK is cryptographically combined with the origin and destination identifiers and returned in a form that can be stored in the CKDS or application storage. In the second step, the partially notarized AKEK is exclusive OR-ed with a binary counter to complete the notarization process. Partial notarization improves performance when you use an AKEK for many cryptographic service messages, each with a different counter. For details of the partial notarization calculations, refer to “ANSI X9.17 Partial Notarization Method” on page 441.

ICSF provides the following callable services to support the ANSI X9.17 key management standard. Except where noted, these callable services have the identical syntax as the Transaction Security System verbs of the same name. With few exceptions, key management applications that use these common callable services, or verbs, can be executed on either system without change. Internal tokens cannot be interchanged; external tokens can be.

Key Generate Callable Service Used to Generate an AKEK

The key generate callable service, described in “Key Generate Callable Service” on page 22, can also be used to generate an AKEK in the operational form. It generates either an 8-byte or 16-byte AKEK and places it in a skeleton key token created by the key token build callable service. The length of the AKEK is determined by the key length keyword specified when building the key token.

ANSI X9.17 EDC Generate Callable Service

This service generates an ANSI X9.17 error detection code on an arbitrary length string.

ANSI X9.17 Key Export Callable Service

This service uses the ANSI X9.17 protocol to export a DATA key or a pair of DATA keys, with or without an AKEK. It also provides the ability to convert a single supplied DATA key or combine two supplied DATA keys into a MAC key.

ANSI X9.17 Key Import Callable Service

This service uses the ANSI X9.17 protocol to import a DATA key or a pair of DATA keys, with or without an AKEK. It also provides the ability to convert a single supplied DATA key or combine two supplied DATA keys into a MAC key. The syntax is identical to the Transaction Security System verb, with the following exceptions:

- Keys cannot be imported directly into the CKDS.

ANSI X9.17 Key Translate Callable Service

This service translates one or two DATA keys or an AKEK from encryption under one AKEK to encryption under another AKEK, using the ANSI X9.17 protocol.

ANSI X9.17 Transport Key Partial Notarize Callable Service

This service preprocesses or partially notarizes an AKEK with origin and destination identifiers. The partially notarized key is supplied to the ANSI X9.17 key export, ANSI X9.17 key import, or ANSI X9.17 key translate callable service to complete the notarization process. The syntax is identical to the Transaction Security System verb except that:

- The callable service does not update the CKDS.

Enciphering and Deciphering Data

The encipher and decipher callable services protect data off the host. ICSF protects sensitive data from disclosure to people who do not have authority to access it. Using algorithms that make it difficult and expensive for an unauthorized user to derive the original clear data within a practical time period assures privacy.

To protect data, ICSF can use the Data Encryption Standard (DES) algorithm to encipher or decipher data or keys. The algorithm is documented in the *Federal Information Processing Standard #46*. You can use the encipher and decipher callable services to encipher and decipher data with encrypted keys. ICSF also supports the CDMF encryption mode. See “System Encryption Algorithm” on page 26 for more information.

The Symmetric Key Encipher and Symmetric Key Decipher callable services are used to encipher and decipher data in an address space or a data space using the cipher block chaining for electronic code book modes. The Advanced Encryption Standard (AES) is supported. AES encryption uses a 128-, 192- or 256-bit key. Only clear keys will be supported. The AES encryption is subject to the same availability restrictions as triple-DES encryption.

Encoding and Decoding Data

The encode and decode callable services perform functions with clear keys. Encode enciphers 8 bytes of data using the electronic code book (ECB) mode of the DES and a clear key. Decode does the inverse of the encode service. These services are available only on a DES-capable system. (See “System Encryption Algorithm” on page 26 for more information.)

Translating Ciphertext

ICSF also provides a ciphertext translate callable service. This service is available only on a DES-capable system. (See “System Encryption Algorithm” on page 26 for more information.) It deciphers encrypted data (ciphertext) under one encryption key and reenciphers it under another key without having the data appear in the clear outside the cryptographic feature. Such a function is useful in a multiple node network, where sensitive data is passed through multiple nodes before it reaches its final destination. Different nodes use different keys in the process. For more information about different nodes, see “Using the Ciphertext Translate Callable Service” on page 39.

The keys cannot be used for the encipher and decipher callable services.

Managing Data Integrity and Message Authentication

To ensure the integrity of transmitted messages and stored data, ICSF provides:
- Message authentication code (MAC)
- Several hashing functions, including modification detection code (MDC), SHA-1, and MD5

(See Chapter 8, “Using Digital Signatures” on page 265 for an alternate method of message authentication using digital signatures.)

The choice of callable service depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender and also the integrity of the data, consider message authentication code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions, such as the modification detection code process.

Message Authentication Code Processing

The process of verifying the integrity and authenticity of transmitted messages is called *message authentication*. Message authentication code (MAC) processing allows you to verify that a message was not altered or a message was not fraudulently introduced onto the system. You can check that a message you have received is the same one sent by the message originator. The message itself may be in clear or encrypted form. The comparison is performed within the cryptographic feature. Since both the sender and receiver share a secret cryptographic key used in the MAC calculation, the MAC comparison also ensures the authenticity of the message.

In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

ICSF provides support for both single-length and double-length MAC generation and MAC verification keys. With the ANSI X9.9-1 single key algorithm, use the single-length MAC and MACVER keys. With the ANSI X9.19 optional double key algorithm, use the double-length DATAM and DATAMV keys available with APAR OW37791.

ICSF provides support for the use of data-encrypting keys in the MAC generation and verification callable services, and also the use of a MAC generation key in the MAC verification callable service. This support permits ICSF MAC services to interface more smoothly with non-CCA key distribution system, including those implementing the ANSI X9.17 protocol.

MAC Generation Callable Service

When a message is sent, an application program can generate an authentication code for it using the MAC generation callable service. The callable service computes the message authentication code using one of the following methods:

- Using the ANSI X9.9-1 single key algorithm, a single-length MAC generation key or data-encrypting key, and the message text.
- Using the ANSI X9.19 optional double key algorithm, a double-length MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

ICSF allows a MAC to be the leftmost 32 or 48 bits of the last block of the ciphertext or the entire last block (64 bits) of the ciphertext. The originator of the message sends the message authentication code with the message text.

MAC Verification Callable Service

When the receiver gets the message, an application program calls the MAC verification callable service. The callable service verifies a MAC by generating another MAC and comparing it with the MAC received with the message. If the two codes are the same, the message sent was the same one received. A return code indicates whether the MACs are the same.

The MAC verification callable service can use either of the following methods to generate the MAC for authentication:

- The ANSI X9.9-1 single key algorithm, a single-length MAC verification or MAC generation key (or a data-encrypting key), and the message text.
- The ANSI X9.19 optional double key algorithm, a double-length MAC verification or MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

The method used to verify the MAC should correspond with the method used to generate the MAC.

Hashing Functions

Hashing functions include one-way hash generation and modification detection code (MDC) processing.

One-Way Hash Generate Callable Service

This service hashes a supplied message. Supported hashing methods include:
- SHA-1[2]
- MD5
- RIPEMD-160

Also supported, through the MDC generation callable service are:
- MDC-2
- MDC-4
- PADMDC-2
- PADMDC-4

MDC Generation Callable Service

The modification detection code (MDC) provides a form of support for data integrity. The MDC allows you to verify that data was not altered during transmission or while in storage. The originator of the data ensures that the MDC is transmitted with integrity to the intended receiver of the data. For instance, the MDC could be published in a reliable source of public information. When the receiver gets the data, an application program can generate an MDC, and compare it with the original MDC value. If the MDC values are equal, the data is accepted as unaltered. If the MDC values differ the data is assumed to be bogus.

In a similar manner, MDCs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

When data is sent, an application program can generate a modification detection code for it using the MDC generation callable service. The callable service computes the modification detection code by encrypting the data using a publicly-known cryptographic one-way function. The MDC is a 128-bit value that is easy to compute for specific data, yet it is hard to find data that will result in a given MDC.

Once an MDC has been established for a file, the MDC generate service can be run at any later time on the file. The resulting MDC can then be compared with the previously established MDC to detect deliberate or inadvertent modification.

Managing Personal Authentication

The process of validating personal identities in a financial transaction system is called *personal authentication*. The personal identification number (PIN) is the basis for verifying the identity of a customer across the financial industry networks. ICSF checks a customer-supplied PIN by verifying it using an algorithm. The financial industry needs functions to generate, translate, and verify PINs. These functions prevent unauthorized disclosures when organizations handle personal identification numbers.

ICSF supports the following algorithms for generating and verifying personal identification numbers:

[2] The Secure Hash Algorithm (SHA) is also called the Secure Hash Standard (SHS), which Federal Information Processing Standard (FIPS) Publication 180 defines.

- IBM 3624
- IBM 3624 PIN offset
- IBM German Bank Pool
- IBM German Bank Pool PIN Offset (GBP-PINO)
- VISA PIN validation value
- Interbank

Note: Interbank is available only on S/390 Enterprise Servers and S/390 Multiprise.

With ICSF, you can translate PIN blocks from one format to another. ICSF supports the following formats:
- ANSI X9.8
- ISO formats 0, 1, 2
- VISA formats 1, 2, 3, 4
- IBM 4704 Encrypting PINPAD format
- IBM 3624 formats
- IBM 3621 formats
- ECI formats 1, 2, 3

With the capability to translate personal identification numbers into different PIN block formats, you can use personal identification numbers on different systems.

Verifying Credit Card Data

The Visa International Service Association (VISA) and MasterCard International, Incorporated have specified a cryptographic method to calculate a value that relates to the personal account number (PAN), the card expiration date, and the service code. The VISA card-verification value (CVV) and the MasterCard card-verification code (CVC) can be encoded on either track 1 or track 2 of a magnetic striped card and are used to detect forged cards. Because most online transactions use track-2, the ICSF callable services generate and verify the CVV[3] by the track-2 method.

The VISA CVV service generate callable service calculates a 1- to 5-byte value through the DES-encryption of the PAN, the card expiration date, and the service code using two data-encrypting keys or two MAC keys. The VISA CVV service verify callable service calculates the CVV by the same method, compares it to the CVV supplied by the application (which reads the credit card's magnetic stripe) in the *CVV_value*, and issues a return code that indicates whether the card is authentic.

Clear PIN Encrypt Callable Service

To format a PIN into a PIN block format and encrypt the results, use the Clear PIN Encrypt callable service. You can also use this service to create an encrypted PIN block for transmission. With the RANDOM keyword, you can have the service generate random PIN numbers. Use of this service requires the optional PCI Cryptographic Coprocessor.

Clear PIN Generate Alternate Callable Service

To generate a clear VISA PIN validation value from an encrypted PIN block, call the clear PIN generate alternate callable service. This service also supports the IBM-PINO algorithm to produce a 3624 offset from a customer selected encrypted PIN.

[3] The VISA CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.

Note: The PIN block must be encrypted under either an input PIN-encrypting key (IPINENC) or output PIN-encrypting key (OPINENC). Using an IPINENC key requires NOCV keys to be enabled in the CKDS. Functions other than VISA PIN validation value generation require the optional PCI Cryptographic Coprocessor.

Clear PIN Generate Callable Service

To generate personal identification numbers, call the Clear PIN generate callable service. Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the callable service generates a clear PIN, a PIN verification value, or an offset. The callable service can only execute in special secure mode, which is described in "Special Secure Mode" on page 9.

Encrypted PIN Generate Callable Service

To generate personal identification numbers, call the Encrypted PIN generation callable service. Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the callable service generates a PIN and using a PIN block format and the PIN encrypting key, formats and encrypts the PIN. Use of this service requires the optional PCI Cryptographic Coprocessor.

Encrypted PIN Translate Callable Service

To translate a PIN from one PIN-encrypting key to another or from one PIN block format to another or both, call the Encrypted PIN translation callable service. You must identify the input PIN-encrypting key that originally enciphers the PIN. You also need to specify the output PIN-encrypting key that you want the callable service to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.

Encrypted PIN Verify Callable Service

To verify a supplied PIN, call the Encrypted PIN verify callable service. You need to specify the supplied enciphered PIN, the PIN-encrypting key that enciphers it, and other relevant data. You must also specify the PIN verification key and PIN verification algorithm. It compares the two personal identification numbers; if they are the same, it verifies the supplied PIN. See Chapter 7, "Financial Services" on page 209 for additional information.

Secure Messaging

The following services will assist applications in encrypting secret information such as clear keys and PIN blocks in a secure message. These services will execute within the secure boundary of the PCI Cryptographic Coprocessor.

The Secure Messaging for Keys (CSNBSKY) callable service encrypts a text block, including a clear key value decrypted from an internal or external DES token.

The Secure Messaging for PINs (CSNBSPN) callable service encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.

Trusted Key Entry (TKE) Support

The Trusted Key Entry (TKE) workstation is an optional feature. It offers an alternative to clear key entry. You can use the TKE workstation to load:

- DES master keys, PKA master keys (see “PKA Master Keys” on page 47), and operational TRANSPORT and PIN keys in a *secure* way.
- SYM-MK and ASYM-MK master keys on the PCI Cryptographic Coprocessor.

You can load keys remotely and for multiple Cryptographic Coprocessor Features and PCI Cryptographic Coprocessors. The TKE workstation eases the administration for using one Cryptographic Coprocessor Feature as a production machine and as a test machine at the same time, while maintaining security and reliability.

The TKE workstation can be used for enabling/disabling access control points for callable services executed on PCI Cryptographic Coprocessors. See Appendix H, “Access Control Points and Callable Services” on page 447 for additional information.

For complete details about the TKE workstation (Version 3 or later), see *z/OS ICSF TKE Workstation User’s Guide 2000*.

Utilities

ICSF provides the following utilities.

Character/Nibble Conversion Callable Services

The character/nibble conversion callable services are utilities that convert a binary string to a character string and vice versa.

Code Conversion Callable Services

The code conversion callable services are utilities that convert EBCDIC data to ASCII data and vice versa.

X9.9 Data Editing Callable Service

The data editing callable service is a utility that edits an ASCII text string according to the editing rules of ANSI X9.9-4.

Typical Sequences of ICSF Callable Services

Sample sequences in which the ICSF callable services might be called are shown in Table 4 on page 35.

Table 4. Combinations of the Callable Services

```
   Combination A (DATA keys only)     Combination B

1. Random number generate          1. Random number generate
2. Clear key import or             2. Secure key import or
   multiple clear key import          multiple secure key import
3. Encipher/decipher               3. Any service
4. Data key export or key export   4. Data key export for DATA keys, or
    (optional step)                   key export in the general case
                                       (optional step)

   Combination C                      Combination D

1. Key generate (OP form only)     1. Key generate (OPEX form)
2. Any service                     2. Any service
3. Key export (optional)

   Combination E                      Combination F

1. Key generate (IM form only)     1. Key generate (IMEX form)
2. Key import                      2. Key import
3. Any service                     3. Any service
4. Key export (optional)

   Combination G                      Combination H

1. Key generate                    1. Key import
2. Key record create               2. Key record create
3. Key record write                3. Key record write
4. Any service (passing label      4. Any service (passing label
   of the key just generated)         of the key just generated)

   Combination I

1. Key token build to create
   key token skeleton
2. Key generate to OP form of
   AKEK using key token skeleton
3. Use AKEK in any ANSI X9.17
   service
```

Notes:

1. An example of “any service” is CSNBENC.
2. These combinations exclude services that can be used on their own; for example, key export or encode, or using key generate to generate an exportable key.
3. These combinations do not show key communication, or the transmission of any output from an ICSF callable service.

The key forms are described in “Key Generate (CSNBKGN)” on page 82.

Key Forms and Types Used in the Key Generate Callable Service

The key generate callable service is the most complex of all the ICSF callable services. This section provides examples of the key forms and key types used in the key generate callable service.

Generating an Operational Key

To generate an operational key, choose one of the following methods:

- **For operational keys,** call the key generate callable service (CSNBKGN). Table 15 on page 91 and Table 16 on page 91 show the key type and key form combinations for a single key and for a key pair.
- **For operational keys,** call the random number generate callable service (CSNBRNG) and specify the *form* parameter as RANDOM. Specify ODD parity for a random number you intend to use as a key. Then pass the generated value to the secure key import callable service (CSNBSKI) with a required key type. The required key type is now in operational form.

  This method requires a cryptographic unit to be in special secure mode. For more information about special secure mode, see “Special Secure Mode” on page 9.
- **For data-encrypting keys,** call the random number generate callable service (CSNBRNG) and specify the *form* parameter as ODD. Then pass the generated value to the clear key import callable service (CSNBCKI) or the multiple clear key import callable service (CSNBCKM). The DATA key type is now in operational form.

You cannot generate a PIN verification (PINVER) key in operational form because the originator of the PIN generation (PINGEN) key generates the PINVER key in exportable form, which is sent to you to be imported.

Generating an Importable Key

To generate an importable key form, call the key generate callable service (CSNBKGN).

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in importable form, obtain it directly by generating a single key. If you want any other key type in importable form, request a key pair where either the first or second key type is importable (IM). Discard the generated key form that you do not need.

Generating an Exportable Key

To generate an exportable key form, call the key generate callable service (CSNBKGN).

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in exportable form, obtain it directly by generating a single key. If you want any other key type in exportable form, request a key pair where either the first or second key type is exportable (EX). Discard the generated key form that you do not need.

Examples of Single-Length Keys in One Form Only

```
Key   Key
Form   1

OP   DATA  Encipher or decipher data. Use data key export or key export
           to send encrypted key to another cryptograpic partner. Then
           communicate the ciphertext.
OP   MAC   MAC generate. Because no MACVER key exists, there is no
           secure communication of the MAC with another cryptographic
           partner.
IM   DATA  Key Import, and then encipher or decipher. Then key export
           to communicate ciphertext and key with another cryptographic
           partner.
EX   DATA  You can send this key to a cryptographic partner, but you
```

can do nothing with it directly. Use it for the key distribution service. The partner could then use key import to get it in operational form, and use it as in OP DATA above.

Examples of OPIM Single-Length, Double-Length, and Triple-Length Keys in Two Forms

The first two letters of the key form indicate the form that key type 1 parameter is in, and the second two letters indicate the form that key type 2 parameter is in.

Key Form	Type 1	Type 2	
OPIM	DATA	DATA	Use the OP form in encipher. Use key export with the OP form to communicate ciphertext and key with another cryptographic partner. Use key import at a later time to use encipher or decipher with the same key again.
OPIM	MAC	MAC	Single-length MAC generation key. Use the OP form in MAC generation. You have no corresponding MACVER key, but you can call the MAC verification service with the MAC key directly. Use the key import callable service and then compute the MAC again using the MAC verification callable service, which comapres the MAC it generates with the MAC supplied with the message and issues a return code indicating whether they compare.

Examples of OPEX Single-Length, Double-Length, and Triple-Length Keys in Two Forms

Key Form	Type 1	Type 2	
OPEX	DATA	DATA	Use the OP form in encipher. Send the EX form and the ciphertext to another cryptographic partner.
OPEX	MAC	MAC	Single-length MAC generation key. Use the OP form in both MAC generation and MAC verification. Send the EX form to a cryptographic partner to be used in the MAC generation or MAC verification services.
OPEX	MAC	MACVER	Single-length MAC generation and MAC verification keys. Use the OP form in MAC generation. Send the EX form to a cryptographic partner where it will be put into key import, and then MAC verification, with the message and MAC that you have also transmitted.
OPEX	PINGEN	PINVER	Use the OP form in Clear PIN generate. Send the EX form to a cryptographic partner where it is put into key import, and then Encrypted PIN verify, along with an IPINENC key.
OPEX	IMPORTER	EXPORTER	Use the OP form in key import, key generate, or secure key import. Send the EX form to a cryptographic partner where it is used in key export, data key export, or key generate, or put in the CKDS.
OPEX	EXPORTER	IMPORTER	Use the OP form in key export, data key export, or key generate. Send the EX form to a cryptographic partner where it is put into the CKDS or used in key import, key generate or secure key import.

When you and your partner have the OPEX IMPORTER EXPORTER, OPEX EXPORTER IMPORTER pairs of keys in “Examples of OPEX Single-Length, Double-Length, and Triple-Length Keys in Two Forms” on page 37 installed, you can start key and data exchange.

Examples of IMEX Single-Length and Double-Length Keys in Two Forms

```
Key    Type    Type
Form   1        2

IMEX   DATA    DATA      Use the key import callable service to import
                         IM form and use the OP form in encipher. Send
                         the EX form to a cryptographic partner.
IMEX  MAC      MACVER    Use the key import callable service to import
                         the IM form and use the OP form in MAC
                         generate. Send the EX form to a cryptographic
                         partner who can verify the MAC.
IMEX IMPORTER  EXPORTER  Use the key import callable service to import
                         the IM form and send the EX form to a
                         cryptographic partner. This establishes a new
                         IMPORTER/EXPORTER key between you and your
                         partner.
IMEX  PINGEN   PINVER    Use the key import callable service to import
                         the IM form and send the EX form to a
                         cryptographic partner. This establishes a new
                         PINGEN/PINVER key between you and your partner.
```

Examples of EXEX Single-Length and Double-Length Keys in Two Forms

For the keys shown in the following list, you are providing key distribution services for other nodes in your network, or other cryptographic partners. Neither key type can be used in your installation.

```
Key    Type    Type
Form    1       2

EXEX  DATA     DATA     Send the first EX form to a cryptographic
EXEX  MAC      MACVER   partner with the corresponding IMPORTER and
EXEX  IMPORTER EXPORTER send the second EX form to another
EXEC  OPINENC  IPINENC  cryptographic partner with the corresponding
                        IMPORTER. This exchange establishes a key
                        between two partners.
```

Generating AKEKs

AKEKs are bidirectional and are OP-form-only keys that can be used in both import and export. Before using the key generate callable service to create an AKEK, you need to use the key token build callable service to create a key token for receiving the AKEK. The steps involved in this process are presented below.

1. Use the key token build callable service with the following parameter values:

Parameter	**Value**
Key_type	AKEK
Rule_array	INTERNAL NO-KEY {SINGLE or DOUBLE-O}

2. Use the key generate callable service with the following parameter values:

Parameter	**Value**
Key_form	OP
Key_type_1	TOKEN

Generated_key_identifier_1
The skeleton key token created in step 1

Using the Ciphertext Translate Callable Service

Note: The ciphertext translate callable service does not work in CDMF-only systems (see “System Encryption Algorithm” on page 26).

This section describes a scenario using the encipher, ciphertext translate, and decipher callable services with four network nodes: A, B, C, and D. You want to send data from your network node A to a destination node D. You cannot communicate directly with node D, and nodes B and C are situated between you. You do not want nodes B and C to decipher your data.

At node A, you use the encipher callable service (CSNBENC or CSNBENC1). Node D uses the decipher callable service (CSNBDEC or CSNBDEC1).

Node B and C will use the ciphertext translate callable service. Consider the keys that are needed to support this process:

1. At your node, generate one key in two forms: OPEX DATA DATAXLAT
2. Send the exportable DATAXLAT key to node B.
3. Node B and C need to share a DATAXLAT key, so generate **a different key** in two forms: EXEX DATAXLAT DATAXLAT.
4. Send the first exportable DATAXLAT key to node B.
5. Send the second exportable DATAXLAT key to node C.
6. Node C and node D need to share a DATAXLAT key and a DATA key. Node D can generate one key in two forms: OPEX DATA DATAXLAT.
7. Node D sends the exportable DATAXLAT key to node C.

The communication process is shown as:

```
Node:       A            B                    C                 D

Callable
Service: Encipher  Ciphertext Translate  Ciphertext Translate   Decipher

Keys:     DATA      DATAXLAT  DATAXLAT    DATAXLAT  DATAXLAT     DATA

Key Pairs: |____ = ____|         |____ = ____|         |____ = ____|
```

Therefore, you need three keys, each in two different forms. You can generate two of the keys at node A, and node D can generate the third key. Note that the key used in the decipher callable service at node D is **not** the same key used in the encipher callable service at node A.

Summary of the DES Callable Services

Table 5 lists the DES callable services described in this book, and their corresponding verbs. The figure also references the chapter that describes the callable service.

Table 5. Summary of ICSF DES Callable Services

Verb	Service Name	Function
Chapter 4, “Managing DES Cryptographic Keys”		

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNBCKI	Clear key import	Imports an 8-byte clear DATA key, enciphers it under the master key, and places the result into an internal key token. CSNBCKI converts the clear key into operational form as a DATA key.
CSNBCVG	Control vector generate	Builds a control vector from keywords specified by the *key_type* and *rule_array* parameters.
CSNBCVT	Control vector translate	Changes the control vector used to encipher an external key.
CSNBCVE	Cryptographic variable encipher	Uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. The plaintext must be a multiple of eight bytes in length.
CSNBDKX	Data key export	Converts a DATA key from operational form into exportable form.
CSNBDKM	Data key import	Imports an encrypted source DES single- or double-length DATA key and creates or updates a target internal key token with the master key enciphered source key.
CSNBDKG	Diversified key generate	Generates a key based upon the key-generating key, the processing method, and the parameter data that is supplied.
CSNBKEX	Key export	Converts any key from operational form into exportable form. (However, this service does not export a key that was marked non-exportable when it was imported.)
CSNBKGN	Key generate	Generates a 64-bit, 128-bit, or 192-bit odd parity key, or a pair of keys; and returns them in encrypted forms (operational, exportable, or importable). CSNBKGN does not produce keys in plaintext.
CSNBKIM	Key import	Converts any key from importable form into operational form.
CSNBKPI	Key part import	Combines the clear key parts of any key type and returns the combined key value in an internal key token or an update to the CKDS.
CSNBKRC	Key record create	Adds a key record containing a key token set to binary zeros to both the in-storage and DASD copies of the CKDS.
CSNBKRD	Key record delete	Deletes a key record from both the in-storage and DASD copies of the CKDS.
CSNBKRR	Key record read	Copies an internal key token from the in-storage copy of the CKDS to application storage.
CSNBKRW	Key record write	Writes an internal key token to the CKDS record specified in the key label parameter. Updates both the in-storage and DASD copies of the CKDS currently in use.

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNBKYT or CSNBKYTX	Key test service	Generates or verifies (depending on keywords in the rule array) a secure verification pattern for keys. CSNBKYT requires the tested key to be in the clear or encrypted under the master key. CSNBKYTX also allows the tested key to be encrypted under a key-encrypting key.
CSNBKTB	Key token build	Builds an internal or external token from the supplied parameters. You can use this callable service to build an internal token for an AKEK for input to the key generate and key part import callable services. You can also use this service to build CCA key tokens for all key types ICSF supports or to update the DES, CDMF, or SYS-ENC markings in a supplied DATA, IMPORTER, or EXPORTER token.
CSNBKTR	Key translate	Uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.
CSNBCKM	Multiple clear key import	Imports a single-, double-, or triple-length clear DATA key, enciphers it under the master key, and places the result into an internal key token. CSNBCKM converts the clear key into operational form as a DATA key.
CSNBSKM	Multiple secure key import	Enciphers a single-, double-, or triple-length clear key under the master key or an input importer key, and places the result into an internal or external key token as any key type. Triple-length keys can only be imported as DATA keys. CSNBSKM executes only in special secure mode.
CSNDPKD	PKA decrypt	Uses an RSA private key to decrypt the RSA-encrypted key value and return the clear key value to the application.
CSNDPKE	PKA encrypt	Encrypts a supplied clear key value under an RSA public key.
CSNBPEX	Prohibit export	Modifies an operational key so that it cannot be exported.
CSNBPEXX	Prohibit export extended	Changes the external token of a key in exportable form so that it can be imported at the receiver node but not exported from that node.
CSNBRNG	Random number generate	Generates an 8-byte random number. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.
CSNBSKI	Secure key import	Enciphers a clear key under the master key, and places the result into an internal or external key token as any key type. CSNBSKI executes only in special secure mode.

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNDSYG	Symmetric key generate	Generates a symmetric DATA key and returns the key in two forms: enciphered under the DES master key or KEK and under a PKA public key.
CSNDSYI	Symmetric key import	Imports a symmetric DATA key enciphered under an RSA public key into operational form enciphered under a DES master key.
CSNDSYX	Symmetric key export	Transfers an application-supplied symmetric key (a DATA key) from encryption under the DES host master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be an ICSF DES internal key token or the label of such a token in the CKDS.
CSNBTCK	Transform CDMF key	Changes a CDMF DATA key in an internal or external token to a transformed shortened DES key.
CSFUDK	User Derived Key	Generates single-length or double-length MAC keys, or updates an existing user derived key.
Chapter 5, “Protecting Data”		
CSNBCTT or CSNBCTT1	Ciphertext translate	Translates the user-supplied ciphertext from one key and enciphers the ciphertext to another key. (This is for DES encryption only.) CSNBCTT requires the ciphertext to reside in the caller’s primary address space. CSNBCTT1 allows the ciphertext to reside in the caller’s primary address space or in an z/OS MVS data space.
CSNBDEC or CSNBDEC1	Decipher	Deciphers data using either the CDMF or the cipher block chaining mode of the DES. (The method depends on the token marking or keyword specification.) The result is called plaintext. CSNBDEC requires the plaintext and ciphertext to reside in the caller’s primary address space. CSNBDEC1 allows the plaintext and ciphertext to reside in the caller’s primary address space or in an z/OS MVS data space.
CSNBDCO	Decode	Decodes an 8-byte string of data using the electronic code book mode of the DES. (This is for DES encryption only.)

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNBENC or CSNBENC1	Encipher	Enciphers data using either the CDMF or the cipher block chaining mode of the DES. (The method depends on the token marking or keyword specification.) The result is called ciphertext. CSNBENC requires the plaintext and ciphertext to reside in the caller's primary address space. CSNBENC1 allows the plaintext and ciphertext to reside in the caller's primary address space or in an z/OS MVS data space.
CSNBECO	Encode	Encodes an 8-byte string of data using the electronic code book mode of the DES. (This is for DES encryption only.)
CSNDSYD	Symmetric key decipher	Deciphers data using the AES algorithm in an address space or a data space using the cipher block chaining or electronic code book modes. AES is the only algorithm supported.
CSNDSYE	Symmetric key encipher	Enciphers data using the AES algorithm in an address space or a data space using the cipher block chaining or electronic code book modes. AES is the only algorithm supported.
Chapter 6, "Verifying Data Integrity and Authenticating Messages"		
CSNBMGN or CSNBMGN1	MAC generate	Generates a 4-, 6-, or 8-byte message authentication code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm. CSNBMGN requires data to reside in the caller's primary address space. CSNBMGN1 allows data to reside in the caller's primary address space or in an z/OS MVS data space.
CSNBMVR or CSNBMVR1	MAC verify	Verifies a 4-, 6-, or 8-byte message authentication code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm and is compared with a user-supplied MAC. CSNBMVR requires data to reside in the caller's primary address space. CSNBMVR1 allows data to reside in the caller's primary address space or in an z/OS MVS data space.

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNBMDG or CSNBMDG1	MDC generate	Generates a 128-bit modification detection code (MDC) for a text string that the application program supplies. CSNBMDG requires data to reside in the caller's primary address space. CSNBMDG1 allows data to reside in the caller's primary address space or in an z/OS MVS data space.
CSNBOWH or CSNBOWH1	One way hash generate	Generates a one-way hash on specified text.
Chapter 7, "Financial Services"		
CSNBCPE	Clear PIN encrypt	Formats a PIN into a PIN block format and encrypts the results.
CSNBPGN	Clear PIN generate	Generates a clear personal identification number (PIN), a PIN verification value (PVV), or an offset using one of the following algorithms: IBM 3624 (IBM-PIN or IBM-PINO) IBM German Bank Pool (GBP-PIN or GBP-PINO) VISA PIN validation value (VISA-PVV) Interbank PIN (INBK-PIN) CSNBPGN executes only in special secure mode.
CSNBCPA	Clear PIN generate alternate	Generates a clear VISA PIN validation value (PVV) from an input encrypted PIN block. The PIN block may have been encrypted under either an input or output PIN encrypting key. The IBM-PINO algorithm is supported to produce a 3624 offset from a customer selected encrypted PIN.
CSNBEPG	Encrypted PIN generate	Generates and formats a PIN and encrypts the PIN block.
CSNBPTR	Encrypted PIN translate	Reenciphers a PIN block from one PIN-encrypting key to another and, optionally, changes the PIN block format.
CSNBPVR	Encrypted PIN verify	Verifies a supplied PIN using one of the following algorithms: IBM 3624 (IBM-PIN or IBM-PINO) IBM German Bank Pool (GBP-PIN or GBP-PINO) VISA PIN validation value (VISA-PVV) Interbank PIN (INBK-PIN)
CSNBSKY	Secure messaging for keys	Encrypts a text block, including a clear key value decrypted from an internal or external DES token.
CSNBSPN	Secure messaging for PINs	Encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.
CSNDSBC	SET block compose	Composes the RSA-OAEP block and the DES-encrypted block in support of the SET protocol.

Table 5. Summary of ICSF DES Callable Services (continued)

Verb	Service Name	Function
CSNDSBD	SET block decompose	Decomposes the RSA-OAEP block and the DES-encrypted block to provide unencrypted data back to the caller.
CSNBCSG	VISA CVV service generate	Generates a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC).
CSNBCSV	VISA CVV service verify	Verifies a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC).
Chapter 10, “Utilities”		
CSNBXBC or CSNBXCB	Character/nibble conversion	Converts a binary string to a character string or vice versa.
CSNBXEA or CSNBXAE	Code conversion	Converts EBCDIC data to ASCII data or vice versa.
CSNB9ED	X9.9 data editing	Edits an ASCII text string according to the editing rules of ANSI X9.9–4.
Chapter 11, “Trusted Key Entry Workstation Interfaces”		
CSFPCI	PCI interface	Puts a request to a specific PCI Cryptographic Coprocessor queue and removes the corresponding response when complete. Only the Trusted Key Entry (TKE) workstation uses this service.
CSFPKSC	PKSC interface	Puts a request to a specific cryptographic module and removes the corresponding response when complete. Only the Trusted Key Entry (TKE) workstation uses this service.
Chapter 12, “Managing Keys According to the ANSI X9.17 Standard”		
CSNAEGN	ANSI X9.17 EDC generate	Generates an ANSI X9.17 error detection code on an arbitrary length string using the special MAC key (x’0123456789ABCDEF’).
CSNAKEX	ANSI X9.17 key export	Uses the ANSI X9.17 protocol to export a DATA key or a pair of DATA keys with or without an AKEK. Supports the export of a CCA IMPORTER or EXPORTER KEK. Converts a single DATA key or combines two DATA keys into a single MAC key.
CSNAKIM	ANSI X9.17 key import	Uses the ANSI X9.17 protocol to import a DATA key or a pair of DATA keys with or without an AKEK. Supports the import of a CCA IMPORTER or EXPORTER KEK. Converts a single DATA key or combines two DATA keys into a single MAC key.
CSNAKTR	ANSI X9.17 key translate	Uses the ANSI X9.17 protocol to translate, in a single service call, either one or two DATA keys or a single KEK from encryption under one AKEK to encryption under another AKEK. Converts a single DATA key or combines two DATA keys into a single MAC key.
CSNATKN	ANSI X9.17 transport key partial notarize	Permits the preprocessing of an AKEK with origin and destination identifiers to create a partially notarized AKEK.

Chapter 3. Introducing PKA Cryptography and Using PKA Callable Services

The preceding section focused on DES cryptography or secret-key cryptography. This is symmetric—senders and receivers use the same key (which must be exchanged securely in advance) to encipher and decipher data. DES functions are synchronous and performed at high speed.

Public key cryptography does not require exchanging a secret key. It is asymmetric—the sender and receiver each have a pair of keys, a public key and a different but corresponding private key. PKA functions are performed in an asynchronous processor; this is much slower than for DES functions.

You can use PKA support to exchange CDMF or DES secret keys securely and to compute digital signatures for authenticating messages to users. You can also use public key cryptography in support of secure electronic transactions over open networks, using SET protocols.

PKA Key Algorithms

Public key cryptography uses a key pair consisting of a public key and a private key. The PKA public key uses one of two algorithms:

- Rivest-Shamir-Adleman (RSA)
- Digital Signature Standard (DSS)

The RSA Algorithm

The RSA algorithm is the most widely used and accepted of the public key algorithms. It uses three quantities to encrypt and decrypt text: a public exponent (PU), a private exponent (PR), and a modulus (M). Given these three and some cleartext data, the algorithm generates ciphertext as follows:

$$\text{ciphertext} = \text{cleartext}^{PU} \quad (\text{modulo } M)$$

Similarly, the following operation recovers cleartext from ciphertext:

$$\text{cleartext} = \text{ciphertext}^{PR} \quad (\text{modulo } M)$$

An RSA key consists of an exponent and a modulus. The private exponent must be secret, but the public exponent and modulus need not be secret.

Digital Signature Standard (DSS)

The U.S. National Institute of Standards and Technology (NIST) defines DSS in Federal Information Processing Standard (FIPS) Publication 186.

PKA Master Keys

PKA master keys protect private keys. On the Cryptographic Coprocessor Feature, there are two PKA master keys; the Signature Master Key (SMK) and the RSA Key Management Master Key (KMMK). The SMK protects PKA private keys used only in digital signature services. The KMMK protects PKA private keys used in digital signature services and in the CDMF and DES DATA key distribution functions. On the PCI Cryptographic Coprocessor, PKA keys are protected by the Asymmetric-Keys Master Key (ASYM-MK).

The ASYM-MK on the PCI Cryptographic Coprocessor is a triple-length key used to encipher and decipher PKA keys. In order for the PCI Cryptographic Coprocessor to function, the hash pattern of the ASYM-MK must match the hash pattern of the SMK on the Cryptographic Coprocessor Feature. The ICSF administrator installs the PKA master keys on the Cryptographic Coprocessor Feature and the ASYM-MK on the PCI Cryptographic Coprocessor by using either the pass phrase initialization routine, the Clear Master Key Entry panels, or the optional Trusted Key Entry (TKE) workstation.

Before services are enabled on the PCI Cryptographic Coprocessor, the following conditions must be met:

- The Symmetric-Keys Master Key (SYM-MK) must be installed on the PCI Cryptographic Coprocessor. It must match the Cryptographic Coprocessor Feature DES master key and match the master key that the CKDS was enciphered with.
- The PKDS is required for OS/390 V2 R9 ICSF and above.
- The PKA master keys (SMK and KMMK) on the Cryptographic Coprocessor Feature must be installed and valid.
- The ASYM-MK PKA master key on the PCI Cryptographic Coprocessor must be installed and valid.
- The hash pattern of the ASYM-MK on the PCI Cryptographic Coprocessor must match the hash pattern of the SMK on the Cryptographic Coprocessor Feature.

On the Cryptographic Coprocessor Feature, operational private keys are protected under two layers of DES encryption. They are encrypted under an Object Protection Key (OPK) that in turn is encrypted under the SMK or KMMK. You dynamically generate the OPK for each private key at import time. ICSF provides a public key data set (PKDS) for the storage of application PKA keys. Although you cannot change PKA master keys dynamically, the PKA Key Token Change callable service can be executed to change a private PKA token (RSA or DSS) from encryption under the old ASYM-MK to encryption under the current ASYM-MK. This service requires a PCI Cryptographic Coprocessor and PKA callable services must be enabled. Private tokens encrypted under the KMMK will only be reenciphered if the KMMK was equal to the SMK. Private tokens in the PKDS are reenciphered after the SMK and ASYM-MK keys are changed by executing the Reencipher PKDS panel option. The reenciphered PKDS is then activated through the Activate PKDS panel option.

PKA Callable Services

The Cryptographic Coprocessor Feature available on S/390 Enterprise Servers, the S/390 Multiprise, and the IBM @server zSeries provides RSA and DSS digital signature functions, key management functions, and DES and CDMF key distribution functions. The S/390 G5 Enterprise Server, S/390 G6 Enterprise Server, and the IBM @server zSeries provide the ability to generate RSA keys on the PCI Cryptographic Coprocessor. ICSF provides application programming interfaces to these functions through callable services. The following PKA callable services perform cryptographic functions:

- PKA key generate
- PKA key import
- PKA key token build
- PKA key token change
- PKA public key extract

- PKDS record create
- PKDS record delete
- PKDS record read
- PKDS record write

Callable Services Supporting Digital Signatures

ICSF provides the following services that support digital signatures.

Digital Signature Generate Callable Service

This service generates a digital signature. This service may use either type. It supports the following methods:

- ANSI X9.30 (DSS)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

The input text must have been previously hashed using the one-way hash generate callable service or the MDC generation service.

Digital Signature Verify Callable Service

This service verifies a digital signature using a PKA public key. (There are two types of PKA public key tokens: RSA and DSS. This service can use either type.) It supports the following methods:

- ANSI X9.30 (DSS)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

The text that is input to this service must be previously hashed using the one-way hash generate callable service or the MDC generation service.

Callable Services for PKA Key Management

ICSF provides the following services for PKA key management.

PKA Key Generate Callable Service

This service generates a PKA internal token for use with the DSS algorithm in digital signature services. You can then use the PKA public key extract callable service to extract a DSS public key token from the internal key token. This service also supports the generation of RSA keys on the PCI Cryptographic Coprocessor for use on the PCI Cryptographic Coprocessor.

Input to the PKA key generate callable service is either a skeleton key token created by the PKA key token build callable service or a valid key token. Upon examination of the input skeleton key token, the PKA key generate service routes the key generation request as follows:

- If the skeleton is for a DSS key token, ICSF routes the request to a Cryptographic Coprocessor Feature.
- If the skeleton is for an RSA key, ICSF routes the request to any available PCI Cryptographic Coprocessor.
- If the skeleton is for a retained RSA key, ICSF routes the request to a PCI Cryptographic Coprocessor where the key is generated and retained for additional security.

PKA Key Import Callable Service

This service imports a PKA private key, which may be RSA or DSS.

The key token to import can be in the clear or encrypted. The PKA key token build utility creates a clear PKA key token. The PKA key generate callable service generates either a clear or an encrypted PKA key token.

PKA Key Token Build Callable Service

The PKA key token build callable service is a utility you can use to create an external PKA key token containing an unenciphered private RSA or DSS key. You can supply this token as input to the PKA key import callable service to obtain an operational internal token containing an enciphered private key. You can also use this service to input a clear unenciphered public RSA or DSS key and return the public key in a token format that other PKA services can use directly.

Use this service to build skeleton key tokens for input to the PKA key generate callable service for creation of RSA keys on the PCI Cryptographic Coprocessor.

PKA Key Token Change Callable Service

The PKA key token change callable service is a utility you can use to change PKA key tokens (RSA and DSS) from encipherment with the old PCI Cryptographic Coprocessor asymmetric-keys master key to encipherment with the current PCI Cryptographic Coprocessor asymmetric-keys master key. This callable service only changes private internal tokens. An active PCI Cryptographic Coprocessor is required and PKA callable services must be enabled.

PKA Public Key Extract Callable Service

This service extracts a PKA public key token from a PKA internal (operational) or external (importable) private key token. It performs no cryptographic verification of the PKA private key token.

Callable Services to Update The Public Key Data Set (PKDS)

The Public Key Data Set (PKDS) is a repository for RSA and DSS public and private keys. An application can store keys in the PKDS and refer to them by label when using any of the callable services which accept public key tokens as input. The PKDS update callable services provide support for creating and writing records to the PKDS and reading and deleting records from the PKDS.

PKDS Record Create Callable Service

This service accepts an RSA or DSS private key token in either external or internal format, or an RSA or DSS public key token and writes a new record to the PKDS. An application can create a null token in the PKDS by specifying a token length of zero. The key label must be unique and the caller must be in task mode and cannot be in SRB mode.

PKDS Record Delete Callable Service

This service deletes a record from the PKDS. An application can specify that the entire record be deleted, or that only the contents of the record be deleted. If only the contents of the record are deleted, the record will still exist in the PKDS but will contain only binary zeros. The key label must be unique and the caller must be in task mode and cannot be in SRB mode.

Note: Retained keys cannot be deleted from the PKDS with this service. See "Retained Key Delete (CSNDRKD)" on page 299 for information on deleting retained keys.

PKDS Record Read Callable Service

This service reads a record from the PKDS and returns the contents of that record to the caller. The key label must be unique and the caller must be in task mode and cannot be in SRB mode.

PKDS Record Write Callable Service

This service accepts an RSA or DSS private key token in either external or internal format, or an RSA or DSS public key token and writes over an existing record in the PKDS. An application can check the PKDS for a null record with the label provided and overwrite this record if it does exist. Alternatively, an application can specify to overwrite a record regardless of the contents of the record. The caller must be in task mode and cannot be in SRB mode.

Note: Retained keys cannot be written to the PKDS with the PKDS Record Write service, nor can a retained key record in the PKDS be overwritten with this service.

Callable Services for Working with Retained Private Keys

Private keys can be generated, retained, and used within the secure boundary of a PCI Cryptographic Coprocessor. Retained keys are generated by the PKA Key Generate (CSNDPKG) callable service. The private key values of retained keys never appear in any form outside the secure boundary. All retained keys have an entry in the PKDS that identifies the PCI Cryptographic Coprocessor where the retained private key is stored. ICSF provides the following callable services to list and delete retained private keys.

In the following situations, the PCI Cryptographic Coprocessor clears the master key registers so that the master key values are not disclosed.

- If the PCI Cryptographic Coprocessor detects tampering (the intrusion latch is tripped), ALL installation data is cleared: master keys, retained keys for all domains, as well as roles and profiles.
- If the PCI Cryptographic Coprocessor detects tampering (the secure boundary of the card is compromised), it self-destructs and can no longer be used.
- If you issue a command from the TKE workstation to zeroize a domain

  This command zeroizes the data specific to a domain: master keys and retained keys.
- If you issue a command from the Support Element panels to zeroize all domains.

  This command zeroizes ALL installation data: master keys, retained keys and access control roles and profiles.

Retained Key Delete Callable Service

The retained key delete callable service deletes a key that has been retained within a PCI Cryptographic Coprocessor and also deletes the record containing the key token from the PKDS.

Retained Key List Callable Service

The retained key list callable service lists the key labels of private keys that are retained within the boundaries of PCI Cryptographic Coprocessors installed on your server.

Callable Services for SET Secure Electronic Transaction

SET is an industry-wide open standard for securing bankcard transactions over open networks. The SET protocol addresses the payment phase of a transaction from the individual, to the merchant, to the acquirer (the merchant's current bankcard processor). It can be used to help ensure the privacy and integrity of real time bankcard payments over the Internet. In addition, with SET in place, everyone in the payment process knows who everyone else is. The card holder, the merchant, and the acquirer can be fully authenticated because the core protocol of SET is based on digital certificates. Each participant in the payment transaction holds a certificate that validates his or her identity. The public key infrastructure allows these digital certificates to be exchanged, checked, and validated for every transaction made over the Internet. The mechanics of this operation are transparent to the application.

Under the SET protocol, every online purchase must be accompanied by a digital certificate which identifies the card-holder to the merchant. The buyer's digital certificate serves as an electronic representation of the buyer's credit card but does not actually show the credit card number to the merchant. Once the merchant's SET application authenticates the buyer's identity, it then decrypts the order information, processes the order, and forwards the still-encrypted payment information to the acquirer for processing. The acquirer's SET application authenticates the buyer's credit card information, identifies the merchant, and arranges settlement. With SET, the Internet becomes a safer, more secure environment for the use of payment cards.

ICSF provides the following callable services that can be used in developing SET applications that make use of the S/390 and IBM @server zSeries cryptographic hardware at the merchant and acquirer payment gateway.

SET Block Compose Callable Service

The SET Block Compose callable service performs DES encryption of data, OAEP-formatting through a series of SHA-1 hashing operations, and the RSA-encryption of the Optimal Asymmetric Encryption Padding (OAEP) block.

SET Block Decompose Callable Service

The SET Block Decompose callable service decrypts both the RSA-encrypted and the DES-encrypted data.

PKA Key Tokens

PKA key tokens contain RSA or DSS private or public keys. Although DES tokens are 64 bytes, PKA tokens are variable length because they contain either RSA or DSS key values, which are variable in length. Consequently, length parameters precede all PKA token parameters. The maximum allowed size is 2500 bytes. PKA key tokens consist of a token header, any required sections, and any optional sections. Optional sections depend on the token type. PKA key tokens can be public or private, and private key tokens can be internal or external. Therefore, there are three basic types of tokens, each of which can contain either RSA or DSS information:

- A public key token
- A private external key token
- A private internal key token

Public key tokens contain only the public key. Private key tokens contain the public and private key pair. Table 6 on page 53 summarizes the sections in each type of token.

Table 6. Summary of PKA Key Token Sections

Section	Public External Key Token	Private External Key Token	Private Internal Key Token
Header	X	X	X
RSA or DSS private key information		X	X
RSA or DSS public key information	X	X	X
Key name (optional)		X	X
Internal information			X

As with DES key tokens, the first byte of a PKA key token contains the token identifier which indicates the type of token.

A first byte of X'1E' indicates an external token with a cleartext public key and optionally a private key that is either in cleartext or enciphered by a transport key-encrypting key. An external key token is in importable key form. It can be sent on the link.

A first byte of X'1F' indicates an internal token with a cleartext public key and a private key that is enciphered by the PKA master key and ready for internal use. An internal key token is in operational key form. A PKA private key token must be in operational form for ICSF to use it. (PKA public key tokens are used directly in the external form.)

Formats for public and private external and internal RSA and DSS key tokens begin in "Format of the RSA Public Key Token" on page 368.

PKA Key Management

You can also generate PKA keys in several ways.

- Using the ICSF PKA key generate callable service.
- Using the Transaction Security System PKA key generate verb, or a comparable product from another vendor.

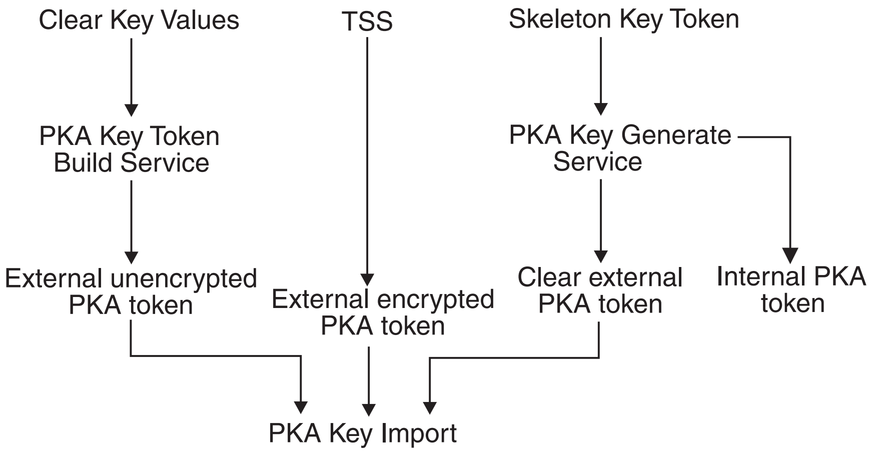


Figure 2. PKA Key Management

If you have a S/390 G5 Enterprise Server, or higher, with a PCI Cryptographic Coprocessor, you can use the ICSF PKA key generate callable service to generate internal and external PKA tokens. You can also generate RSA keys on another system. To input a clear RSA key to ICSF, create the token with the PKA key token build callable service and import it using the PKA key import callable service. To input an encrypted RSA key, generate the key on the Transaction Security System and import it using the PKA key import callable service.

In either case, use the PKA key token build callable service to create a skeleton key token as input (see “PKA Key Token Build (CSNDPKB)” on page 279).

You can generate DSS keys on another system or on ICSF. You need to supply DSS network quantities to the PKA key generate callable service. If you generate DSS keys on another system, you can import them the same way as RSA keys. If you generate a DSS key on ICSF, you can never export it. You can use it on another ICSF host only if the same PKA master keys are installed on both systems.

The PKA key import callable service uses the clear token from the PKA key token build service or a clear or encrypted token from the Transaction Security System to securely import the key token into operational form for ICSF to use. ICSF does not permit the export of the imported PKA key.

The PKA public key extract callable service builds a public key token from a private key token.

Application RSA and DSS public and private keys can be stored in the public key data set (PKDS), a VSAM data set.

Invocation Requirements

The following services have the restriction: The caller must be in task mode, not SRB mode. For all of these, the caller can be in cross-memory mode.

- PKA key import
- Digital signature generate
- Digital signature verify
- Symmetric key export
- Symmetric key import
- Symmetric key generate
- Retained key list
- Retained key delete
- PKA key generate
- SET block compose
- SET block decompose
- PKA encrypt
- PKA decrypt

Security and Integrity of the Token

PKA private key tokens may optionally have a 64-byte *private_key_name* field. If *private_key_name* exists, ICSF uses RACHECK to verify it before using the token in a callable service. For additional security, the processor also validates the entire private key token.

Key Identifier for PKA Key Token

A *key identifier* for a PKA key token is a variable length (maximum allowed size is 2500 bytes) area that contains one of the following:

- **Key label** identifies keys that are in the PKDS. Ask your ICSF administrator for the key labels that you can use.
- **Key token** can be either an internal key token, an external key token, or a null key token. Key tokens are generated by an application (for example, using the PKA key generate callable service), or received from another system that can produce external key tokens.

  An **internal key token** can be used only on ICSF, because a PKA master key encrypts the key value. Internal key tokens contain keys in operational form only.

  An **external key token** can be exchanged with other systems because a transport key that is shared with the other system encrypts the key value. External key tokens contain keys in either exportable or importable form.

  A **null key token** consists of 8 bytes of binary zeros. The PKDS Record Create service can be used to write a null token to the PKDS. This PKDS record can subsequently be identified as the target token for the PKA key import or PKA key generate service.

The term *key identifier* is used when a parameter could be one of the above items and to indicate that different inputs are possible. For example, you may want to specify a specific parameter as either an internal key token or a key label. The key label is, in effect, an indirect reference to a stored internal key token.

Key Label

If the first byte of the key identifier is greater than X'40', the field is considered to be holding a **key label**. The contents of a key label are interpreted as a pointer to a public key data set (PKDS) key entry. The key label is an indirect reference to an internal key token.

A key label is specified on callable services with the *key_identifier* parameter as a 64-byte character string, left-justified, and padded on the right with blanks. In most cases, the callable service does not check the syntax of the key label beyond the first byte. One exception is the key record create callable service which enforces the KGUP rules for key labels unless syntax checking is bypassed by a preprocessing exit.

A key label has the following form:

Offset	Length	Data
00-63	64	Key label name

Key Token

A key token is a variable length (maximum allowed size is 2500 bytes) field composed of key value and control information. PKA keys can be either public or private RSA or DSS keys. Each key token can be either an internal key token (the first byte of the key identifier is X'1F'), an external key token (the first byte of the key identifier is X'1E'), or a null PKA private key token (the first byte of the key identifier is X'00'). The following is a list of private key section identifiers for internal and external private RSA key tokens:

Table 7. Internal and External Private RSA Key Token Section Identifiers

Key token	Section identifier
RSA Private Key Token 1024 Modulus-Exponent External Form	X'02'
RSA Private Key Token 2048 Chinese Remainder Theorem External Form	X'08'
RSA Private Key Token 1024 Modulus-Exponent Internal Form (Cryptographic Coprocessor Feature)	X'02'
RSA Private Key Token 1024 Modulus-Exponent Internal Form (PCI Cryptographic Coprocessor)	X'06'
RSA Private Key Token 2048 Chinese Remainder Theorem Internal Form	X'08'

See Appendix B, “Key Token Formats” on page 365 for descriptions of the PKA key tokens.

An internal key token is a token that can be used only on the ICSF system that created it (or another ICSF system with the same PKA master key). It contains a key that is encrypted under the PKA master key.

An application obtains an internal key token by using one of the callable services such as those listed below. The callable services are described in detail in Chapter 9, “Managing PKA Cryptographic Keys”.

- PKA key generate

The PKA Key Token Change callable service can reecipher private internal tokens from encryption under the old ASYM-MK to encryption under the current ASYM-MK. PKDS Reencipher/Activate options are available to reencipher RSA and DSS internal tokens in the PKDS after the SMK/ASYM-MK keys are changed.

PKA master keys may not be changed dynamically.

For debugging information, see Appendix B, “Key Token Formats” for the format of an internal key token.

If the first byte of the key identifier is X'1E', the key identifier is interpreted as an **external key token**. An external PKA key token contains key (possibly encrypted) and control information. By using the external key token, you can exchange keys between systems.

An application obtains the external key token by using one of the callable services such as those listed below. They are described in detail in Chapter 9, “Managing PKA Cryptographic Keys”.

- PKA public key extract
- PKA key import
- PKA key token build
- PKA key generate

For debugging information, see Appendix B, “Key Token Formats” for the format of an external key token.

If the first byte of the key identifier is X'00', the key identifier is interpreted as a **null key token**. Use the null key token to produce external key tokens.

For debugging information, see Appendix B, “Key Token Formats” for the format of a null key token.

The Transaction Security System and ICSF Portability

The Transaction Security System PKA verbs from releases prior to 1996 can run only on the Transaction Security System. The PKA96 release of the Transaction Security System PKA verbs generally runs on ICSF without change. As with DES cryptography, you cannot interchange internal PKA tokens but can interchange external tokens.

Summary of the PKA Callable Services

Table 8 lists the PKA callable services, described in this book, and their corresponding verbs. (The PKA services start with CSNDxxx and have corresponding CSFxxx names.) This table also references the chapter that describes the callable service.

Table 8. Summary of PKA Callable Services

Verb	Service Name	Function
Chapter 7, “Financial Services”		
CSNDSBC	SET block compose	Composes the RSA-OAEP block and the DES-encrypted block in support of the SET protocol.
CSNDSBD	SET block decompose	Decomposes the RSA-OAEP block and the DES-encrypted block to provide unencrypted data back to the caller.
Chapter 8, “Using Digital Signatures”		
CSNDDSG	Digital signature generate	Generates a digital signature using a PKA private key supporting RSA and DSS algorithms.
CSNDDSV	Digital signature verify	Verifies a digital signature using a PKA public key supporting RSA and DSS algorithms.
Chapter 9, “Managing PKA Cryptographic Keys”		
CSNDPKG	PKA key generate	Generates a DSS internal token for use in digital signature services and RSA keys for use on the PCI Cryptographic Coprocessor.
CSNDPKI	PKA key import	Imports a PKA key token containing either a clear PKA key or a PKA key enciphered under a limited authority IMP-PKA KEK.
CSNDPKB	PKA key token build	Creates an external PKA key token containing a clear private RSA or DSS key. Using this token as input to the PKA key import callable service returns an operational internal token containing an enciphered private key. Using CSNDPKB on a clear public RSA or DSS key, returns the public key in a token format that other PKA services can directly use. CSNDPKB can also be used to create a skeleton token for input to the PKA Key Generate service for the generation of an internal DSS or RSA key token.
CSNDKTC	PKA key token change	Changes PKA key tokens (RSA and DSS) from encipherment with the old PCI Cryptographic Coprocessor asymmetric-keys master key to encipherment with the current PCI Cryptographic Coprocessor asymmetric-keys master key. This callable service only changes private internal tokens.

Table 8. Summary of PKA Callable Services (continued)

Verb	Service Name	Function
CSNDPKX	PKA public key extract	Extracts a PKA public key token from a supplied PKA internal or external private key token. Performs no cryptographic verification of the PKA private token.
CSNDKRC	PKDS record create	Writes a new record to the PKDS.
CSNDKRD	PKDS record delete	Delete a record from the PKDS.
CSNDKRR	PKDS record read	Read a record from the PKDS and return the contents of that record.
CSNDKRW	PKDS record write	Write over an existing record in the PKDS.
CSNDRKL	Retained key list	Lists key labels of keys that have been retained within all currently active PCI Cryptographic Coprocessors.
CSNDRKD	Retained key delete	Deletes a key that has been retained within the PCI Cryptographic Coprocessor.

Part 2. CCA Callable Services

This part of the book introduces DES and PKA callable services.

Chapter 4. Managing DES Cryptographic Keys

This chapter describes the callable services that generate and maintain cryptographic keys.

Using ICSF, you can generate keys using either the key generator utility program or the key generate callable service. ICSF provides a number of callable services to assist you in managing and distributing keys and maintaining the cryptographic key data set (CKDS).

This chapter describes the following callable services:

Clear Key Import (CSNBCKI)

Use the clear key import callable service to import a clear DATA key that is to be used to encipher or decipher data. This callable service can import only DATA keys. Clear key import accepts an 8-byte clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token. This service marks this internal key token CDMF or DES, according to the system's default encryption algorithm, unless token copying overrides this.

If the clear key value does not have odd parity in the low-order bit of each byte, the service returns a warning value in the *reason_code* parameter. The callable service does not adjust the parity of the key.

Clear Key Import (CSNBCKI)

Note: To import 16-byte or 24-byte DATA keys, use the multiple clear key import callable service that is described in “Multiple Clear Key Import (CSNBCKM)” on page 120.

Format

```
CALL CSNBCKI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            clear_key,
            key_identifier )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

clear_key

Direction: Input Type: String

The *clear_key* specifies the 8-byte clear key value to import.

key_identifier

Direction: Input/Output Type: String

A 64-byte string that is to receive the internal key token. “Key Identifier for Key Token” on page 7 describes the internal key token. If this parameter contains a valid internal key token for a DATA key, clear key import propagates the data encryption algorithm bits to the imported key token. Otherwise, this callable service marks the key token according to the system's default algorithm.

Usage Note

This service produces an internal DATA token with a control vector which is usable on the Cryptographic Coprocessor Feature. If a valid internal token is supplied as input to the service in the *key_identifier* field, that token’s control vector will not be used in the encryption of the clear key value.

Control Vector Generate (CSNBCVG)

The Control Vector Generate callable service builds a control vector from keywords specified by the *key_type* and *rule_array* parameters.

Format

```
CALL CSNBCVG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_type,
             rule_array_count,
             rule_array,
             reserved,
             control_vector )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_type

Direction: Input Type: String

A string variable containing a keyword for the key type. The keyword is 8 bytes in length, left justified, and padded on the right with space characters. It is taken from the following list:

```
CIPHER
CVARDEC
CVARENC
CVARPINE
CVARXCVL
CVARXCVR
DATA
DATAC
DATAM
DATAMV
DECIPHER
DKYGENKY
ENCIPHER
EXPORTER
IKEYXLAT
IMPORTER
IPINENC
KEYGENKY
MAC
MACVER
OKEYXLAT
OPINENC
PINGEN
PINVER
SECMSG
```

Note: CLR8-ENC must be coded in *rule_array* when the KEYGENKY key type is coded. When the SECMSG *key_type* is coded, either SMKEY or SMPIN must be specified in the *rule_array*.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter.

rule_array

Direction: Input Type: Character String

Keywords that provide control information to the callable service. Each keyword is left justified in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage. “Key Token Build (CSNBKTB)” on page 109 illustrates the key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build callable services to create a control vector. The rule array keywords are shown below.

```
ANY
CLR8-ENC
CPINENC
CPINGEN
CPINGENA
DALL
DATA
DDATA
DEXP
DIMP
DKYL0
DKYL1
DKYL2
DKYL3
DKYL4
DKYL5
DKYL6
DKYL7
DMAC
DMKEY
DMPIN
DMV
DOUBLE
DPVR
EPINGEN
EPINVER
EXEX
EXPORT
GBP-PIN
GBP-PINO
IBM-PIN
IBM-PINO
IMEX
IMIM
IMPORT
INBK-PIN
KEY-PART
KEYLN8
KEYLN16
LMTD-KEK
MIXED
NO-SPEC
NO-XPORT
NOOFFSET
NOT-KEK
OPEX
OPIM
PIN
REFORMAT
SINGLE
SMKEY
SMPIN
TRANSLAT
UKPT
VISA-PVV
XLATE
XPORT-OK
```

Note: CLR8-ENC must be coded in *rule_array* when the KEYGENKY key type is coded. When the SECMSG *key_type* is coded, either SMKEY or SMPIN must be specified in the *rule_array*.

reserved

Direction: Input Type: String

The *reserved* parameter must be a variable of 8 bytes of X'00'.

control_vector

Direction: Output Type: String

A 16-byte string variable in application storage where the service returns the generated control vector.

Usage Notes

SAF will be invoked to check authorization to use the Control Vector Generate service.

See Table 22 on page 116 for an illustration of key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build callable services to create a control vector.

Control Vector Translate (CSNBCVT)

The Control Vector Translate callable service changes the control vector used to encipher an external key.

ICSF routes the Control Vector Translate request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails. See “Changing Control Vectors with the Control Vector Translate Callable Service” on page 393 for additional information about this service.

Format

```
CALL CSNBCVT(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            KEK_key_identifier,
            source_key_token,
            array_key_left,
            mask_array_left,
            array_key_right,
            mask_array_right,
            rule_array_count,
            rule_array,
            target_key_token )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

KEK_key_identifier

Direction: Input/Output Type: String

The 64-byte string variable containing an internal key token or the key label of an internal key token record containing the key-encrypting key. The control vector in the internal key token must specify the key type of IMPORTER, EXPORTER, IKEYXLAT, or OKEYXLAT.

source_key_token

Direction: Input Type: String

A 64-byte string variable containing the external key token with the key and control vector to be processed.

array_key_left

Direction: Input/Output Type: String

A 64-byte string variable containing an internal key token or a key label of an internal key token record that deciphers the left mask array. The internal key token must contain a control vector specifying a CVARXCVL key type.

mask_array_left

Direction: Input Type: String

A string of seven 8-byte elements containing the mask array enciphered under the left array key.

array_key_right

Direction: Input/Ouput Type: String

A 64-byte string variable containing an internal key token or a key label of an internal key token record that deciphers the right mask array. The internal key token must contain a control vector specifying a CVARXCVR key type.

mask_array_right

Direction: Input Type: String

A string of seven 8-byte elements containing the mask array enciphered under the right array key.

rule_array_count

Direction: Input Type: Integer

An integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero, one, or two for this service. If the rule array count is zero, the default keywords ADJUST and LEFT are used.

rule_array

Direction: Input Type: Character String

The *rule_array* parameter is an array of keywords. The keywords are 8 bytes in length, and must be left-justified and padded on the right with space characters. The rule_array keywords are shown below.

Table 9. Keywords for Control Vector Translate

Keyword	Meaning
Parity Adjustment Rule (optional)	
ADJUST	Ensures that all target key bytes have odd parity. This is the default.
NOADJUST	Prevents the parity of the target being altered.
Key-portion Rule (optional)	
BOTH	Causes both halves of a 16-byte source key to be processed with the result placed into corresponding halves of the target key. When you use the BOTH keyword, the mask array must be able to validate the translation of both halves.
LEFT	Causes an 8-byte source key, or the left half of a 16-byte source key, to be processed with the result placed into both halves of the target key. This is the default.

Table 9. Keywords for Control Vector Translate (continued)

Keyword	Meaning
RIGHT	Causes the right half of a 16-byte source key to be processed with the result placed into the right half of the target key. The left half is copied unchanged (still enciphered) from the source key.
SINGLE	Causes the left half of the source key to be processed with the result placed into the left half of the target key token. The right half of the target key is unchanged.

target_key_token

Direction: Input/Output Type: String

A 64-byte string variable containing an external key token with the new control vector. This key token contains the key halves with the new control vector.

Restriction

The caller must be in task mode, not in SRB mode.

Usage Notes

SAF will be invoked to check authorization to use the Control Vector Translate service and any key labels specified as input.

If *KEK_key_identifier* is a label of an IMPORTER or EXPORTER key, the label must be unique in the CKDS.

Cryptographic Variable Encipher (CSNBCVE)

The Cryptographic Variable Encipher callable service uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. You can use this service to prepare a mask array for the Control Vector Translate service. The plaintext must be a multiple of eight bytes in length.

ICSF routes the cryptographic variable encipher request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBCVE(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            c-variable_encrypting_key_identifier,
            text_length,
            plaintext,
            initialization_vector,
            ciphertext )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

c-variable_encrypting_key_identifier

Direction: Input/Output Type: String

The 64-byte string variable containing an internal key or a key label of an internal key token record in the CKDS. The internal key must contain a control vector that specifies a CVARENC key type.

text_length

Direction: Input Type: Integer

An integer variable containing the length of the plaintext and the returned ciphertext.

plaintext

Direction: Input Type: String

A string of length 8 to 256 bytes which contains the plaintext.

initialization_vector

Direction: Input	Type: String

A string variable containing the 8-byte initialization vector that the service uses in encrypting the plaintext.

ciphertext

Direction: Output	Type: String

The field which receives the ciphertext. The length of this field is the same as the length of the plaintext.

Restrictions

- The text length must be a multiple of 8 bytes.
- The maximum length of text that the security server can process is 256 bytes.
- The caller must be in task mode, not in SRB mode.

Usage Note

SAF will be invoked to check authorization to use the Cryptographic Variable Encipher service and the key label if specified in the *c-variable_encrypting_key_identifier* parameter.

Data Key Export (CSNBDKX)

Use the data key export callable service to reencipher a data-encrypting key (key type of DATA only) from encryption under the master key to encryption under an exporter key-encrypting key. The reenciphered key is in a form suitable for export to another system.

ICSF examines the data encryption algorithm bits on the exporter key-encrypting key and DATA key for consistency. It does not export a CDMF key under a DES-marked key-encrypting key or a DES key under a CDMF-marked key-encrypting key. These checks are not enforced when the service is executed on a PCI Cryptographic Coprocessor. ICSF does not propagate the data encryption marking on the operational key to the external token.

The data key export service generates a key token with the same key length as the input token's key.

ICSF routes the Data Key Export request to a PCI Cryptographic Coprocessor if the control vector of the *exporter_key_identifier* cannot be processed on the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Format

```
CALL CSNBDKX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             source_key_identifier,
             exporter_key_identifier,
             target_key_identifier )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

source_key_identifier

Direction: Input/Output	Type: String

A 64-byte string for an internal key token or label that contains a data-encrypting key to be reenciphered. The data-encrypting key is encrypted under the master key.

exporter_key_identifier

Direction: Input/Output	Type: String

A 64-byte string for an internal key token or key label that contains the exporter *key_encrypting* key. The data-encrypting key above will be encrypted under this exporter *key_encrypting* key.

target_key_identifier

Direction: Input/Output Type: String

A 64-byte field that is to receive the external key token, which contains the reenciphered key that has been exported. The reenciphered key can now be exchanged with another cryptographic system.

Usage Note

You cannot export a CDMF key under a DES-marked KEK or a DES key under a CDMF-marked KEK. The service fails with a return code of 8 and reason code of 10120. These checks are not enforced when the service is executed on a PCI Cryptographic Coprocessor.

Data Key Import (CSNBDKM)

Use the data key import callable service to import an encrypted source DES single-length or double-length DATA key and create or update a target internal key token with the master key enciphered source key. ICSF routes the Data Key Import request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBDKM(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            source_key_token,
            importer_key_identifier,
            target_key_identifier)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

source_key_token

Direction: Input   Type: String

A 64-byte string variable containing the source key to be imported. The source key must be an external key. The external key token must indicate that a control vector is present; however, the control vector is usually valued at zero. A double-length key that should result in a default DATA control vector must be specified in a version X'01' external key token. Otherwise, both single and double-length keys are presented in a version X'00' key token. Alternatively, the encrypted data key can be provided at offset 16 in an otherwise all X'00' key token. The service will process this token format as a DATA key encrypted by the importer key and a null (all zero) control vector.

importer_key_identifier

Direction: Input/Output   Type: String

A 64-byte string variable containing the (IMPORTER) transport key or key label of the transport key used to decipher the source key.

target_key_identifier

Direction: Output   Type: String

A 64-byte string variable containing a null key token or an internal key token. The key token receives the imported key.

Restriction

The caller must be in task mode, not in SRB mode.

Usage Notes

SAF will be invoked to check authorization to use the Data Key Import service and the label of the *importer_key_identifier*.

This service does not adjust the key parity of the source key.

CDMF/DES token markings will be ignored.

Diversified Key Generate (CSNBDKG)

The diversified key generate service generates a key based on the key-generating key, the processing method, and the parameter supplied. The control vector of the key-generating key also determines the type of target key that can be generated.

To use this service, specify the following:

- The rule array keyword to select the diversification process.
- The operational key-generating key from which the diversified keys are generated. The control vector associated with this key restricts the use of this key to the key generation process. This control vector also restricts the type of key that can be generated.
- The data and length of data used in the diversification process.
- The generated-key may be an internal token or a skeleton token containing the desired CV of the generated-key. The generated key CV must be one that is permitted by the processing method and the key-generating key. The generated-key will be returned in this parameter.

This service generates diversified keys as follows:

- Determines if it can support the process specified in rule array.
- Recovers the key-generating key and checks the key-generating key class and the specified usage of the key-generating key.
- Determines that the control vector in the generated-key token is permissible for the specified processing method.
- Determines that the control vector in the generated-key token is permissible by the control vector of the key-generating key.
- Determines the required data length from the processing method and the generated-key CV. Validates the *data_length*.
- Generates the key appropriate to the specific processing method. Adjusts parity of the key to odd. Creates the internal token and returns the generated diversified key.

Format

```
CALL CSNBDKG(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            generating_key_identifier,
            data_length,
            data,
            key_identifier,
            generated_key_identifier)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The only valid value is 1.

rule_array

Direction: Input	Type: String

The keyword that provides control information to the callable service. The processing method is the algorithm used to create the generated key. The keyword is left justified and padded on the right with blanks.

Table 10. Rule Array Keywords for Diversified Key Generate

Keyword	Meaning
Processing Method for generating or updating diversified keys (required)	
CLR8-ENC	Specifies that 8-bytes of clear data shall be multiply encrypted with the generating key. The *generating_key_identifier* must be a KEYGENKY key type with bit 19 of the control vector set to 1. The control vector in *generated_key_identifier* must specify a single-length key. The key type may be DATA, MAC, or MACVER. **Note:** CIPHER class keys are not supported.

Table 10. Rule Array Keywords for Diversified Key Generate (continued)

Keyword	Meaning
TDES-DEC	Data supplied may be 8 or 16 bytes of clear data. If the *generated_key_identifier* specifies a single length key, then 8-bytes of data is TDES decrypted under the *generating_key_identifier*. If the *generated_key_identifier* specifies a double length key, then 16-bytes of data is TDES ECB mode decrypted under the *generating_key_identifier*. No formating of data is done before encryption. The *generating_key_identifier* must be a DKYGENKY key type, with appropriate usage bits for the desired generated key.
TDES-ENC	Data supplied may be 8 or 16 bytes of clear data. If the *generated_key_identifier* specifies a single length key, then 8-bytes of data is TDES encrypted under the *generating_key_identifier*. If the *generated_key_identifier* specifies a double length key, then 16-bytes of data is TDES ECB mode encrypted under the *generating_key_identifier*. No formatting of data is done before encryption. The *generating_key_identifier* must be a DKYGENKY key type, with appropriate usage bits for the desired generated key. The *generated_key_identifier* may be a single or double length key with a CV that is permitted by the *generating_key_identifier*.
Processing Method for updating a diversified key	
SESS-XOR	Specifies the VISA method for session key generation. Data supplied may be 8 or 16 bytes of data depending on whether the *generating_key_identifier* is a single or double length key. The 8 or 16 bytes of data is XORed with the clear value of the *generating_key_identifier*. The *generated_key_identifier* has the same control vector as the *generating_key_identifier*. The *generating_key_identifier* may be DATA/DATAC, MAC/DATAM or MACVER/DATAMV key types.

generating_key_identifier

Direction: Input/Output Type: String

The label or internal token of a key generating key. The type of key-generating key depends on the processing method.

data_length

Direction: Input Type: Integer

The length of the *data* parameter that follows. Length depends on the processing method and the generated key.

data

Direction: Input Type: String

Data input to the diversified key or session key generation process. Data depends on the processing method and the *generated_key_identifier*.

key_identifier

Direction: Input/Output	Type: String

This parameter is currently not used. It must be a 64-byte null token.

generated_key_identifier

Direction: Input/Output	Type: String

The internal token of an operational key, a skeleton token containing the control vector of the key to be generated, or a null token. A null token can be supplied if the *generated_key_identifier* will be a DKYGENKY with a CV derived from the *generating_key_identifier*. A skeleton token or internal token is required when *generated_key_identifier* will not be a DKYGENKY key type or the processing method is not SESS-XOR. For SESS-XOR, this must be a null token. On output, this parameter contains the generated key.

Restrictions

- This service requires at least one PCI Cryptographic Coprocessor to be installed and active.
- The caller must be in task mode and not in SRB mode.

Usage Notes

SAF will be invoked to check authorization to use the Diversified Key Generate service and any key labels specified as input.

Key Export (CSNBKEX)

Use the key export callable service to reencipher any type of key (except an AKEK or an IMP-PKA) from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key. The reenciphered key can be exported to another system.

ICSF examines the data encryption algorithm bits on the exporter key-encrypting key and the key being exported for consistency. It does not export a CDMF key under a DES-marked key-encrypting key or a DES key under a CDMF-marked key-encrypting key. These checks are not enforced when the service is executed on a PCI Cryptographic Coprocessor. ICSF does not propagate the data encryption marking on the operational key to the external token.

If the key to be exported is a DATA key, the key export service generates a key token with the same key length as the input token's key.

This service supports the no-export bit that the prohibit export service sets in the internal token.

ICSF routes the Key Export request to a PCI Cryptographic Coprocessor if the control vector of the *exporter_key_identifier* cannot be processed on the Cryptographic Coprocessor Feature. The service will also be routed to a PCI Cryptographic Coprocessor if the key type specified is one of the following:

DECIPHER, ENCIPHER, IKEYXLAT, OKEYXLAT or CIPHER. If no PCI Cryptographic Coprocessor is online in this case, the request fails. If the key type is MACD, the key export service will be processed on a Cryptographic Coprocessor Feature. When TOKEN is specified for the *key_type*, it is also routed to the PCI Cryptographic Coprocessor. If no PCI Cryptographic Coprocessor is available, and TOKEN was specified, the CV is checked to determine if it can be processed on the Cryptographic Coprocessor Feature. If it can, the service is executed. If not, the request fails.

Format

```
CALL CSNBKEX(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_type,
            source_key_identifier,
            exporter_key_identifier,
            target_key_identifier )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_type

Direction: Input Type: Character string

The parameter is an 8-byte field that contains either a key type value or the keyword TOKEN. The keyword is left-justified and padded on the right with blanks. The key types you can specify are listed in Table 11.

For a double-length MAC key with a key type of DATAM, the service uses the data compatibility control vector to create an external token. For a double-length MAC key with a key type of MACD, the service uses the single-length control vector for both the left and right half of the key to create an external token (MAC ‖ MAC). For a table of control vectors, refer to “Control Vector Table” on page 383.

To export a double-length MAC generation key, use a key type of either DATAM or TOKEN. To export a double-length MAC verification key, use a key type of either DATAMV or TOKEN.

If the key type is TOKEN, ICSF determines the key type from the control vector (CV) field in the internal key token provided in the *source_key_identifier* parameter. If the control vector is invalid on the Cryptographic Coprocessor Feature, the key export request will be routed to the PCI Cryptographic Coprocessor.

Table 11. Key Type Values for the Key Export Callable Service

Key Type	**Meaning**
CIPHER	Used only to encrypt or decrypt data. Possible key lengths are single-, or double-length. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
DATA	Data encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAC	Used to specify a DATA-class key that will perform in the Encipher and Decipher callable services, but not in the MAC Generate or MAC Verify callable services. This is a double-length key.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
DECIPHER	Used only to decrypt data. Possible key lengths are single-, or double-length. DECIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
ENCIPHER	Used only to encrypt data. Possible key lengths are single-, or double-length. ENCIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
EXPORTER	Exporter key-encrypting key. Use this double-length key (including a DATA key) from the operational form into exportable form.

Table 11. Key Type Values for the Key Export Callable Service (continued)

Key Type	Meaning
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IPINENC	Double-length input PIN-encrypting key. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate, Encrypted PIN verify, and Clear PIN Generate Alternate services. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an IPINENC key.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACD	Double-length MAC generation and verification key. ICSF continues to support this key type for compatibility with ICSF/MVS Version 2 Release 1.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Output PIN-encrypting key. Use this double-length output key to translate PINs. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.

source_key_identifier

Direction: Input　　　　Type: String

A 64-byte string of the internal key token that contains the key to be reenciphered. This parameter must identify an internal key token in application storage, or a label of an existing key in the cryptographic key data set.

If you supply TOKEN for the *key_type* parameter, ICSF looks at the control vector in the internal key token and determines the key type from this information. If you supply TOKEN for the *key_type* parameter and supply a label for this parameter, the label must be unique in the cryptographic key data set.

exporter_key_identifier

Direction: Input/Output Type: String

A 64-byte string of the internal key token or key label that contains the exporter key-encrypting key. This parameter must identify an internal key token in application storage, or a label of an existing key in the cryptographic key data set.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key. For example, the key has been installed in the cryptographic key data set through the key generator utility program or the key entry hardware using the NOCV parameter; or you are passing the key-encrypting key in the internal key token with the NOCV bit on and your program is running in supervisor state or in key 0-7.

Control vectors are explained in “Control Vector” on page 16 and the NOCV bit is shown in Table 97 on page 365.

target_key_identifier

Direction: Input/Output Type: String

The 64-byte field external key token that contains the reenciphered key. The reenciphered key can be exchanged with another cryptographic system.

Usage Notes

You cannot export a CDMF key under a DES-marked KEK or a DES key under a CDMF-marked KEK. The service fails with a return code of 8 and reason code of 10120. These checks are not enforced when the service is executed on a PCI Cryptographic Coprocessor.

This service supports the no-export bit that the key import service sets in the internal token. (A return code of 8 and reason code of 10124 indicates a key cannot be exported.)

For key export, you can use the following combinations of parameters:

- A valid key type in the *key_type* parameter and an internal key token in the *source_key_identifier* parameter. The key type must be equivalent to the control vector specified in the internal key token.
- A *key_type* parameter of TOKEN and an internal key token in the *source_key_identifier* parameter. With *key_type* of TOKEN, the *source_key_identifier* can also be a label. The key type is extracted from the control vector contained in the internal key token.
- A valid key type in the *key_type* parameter, and a label in the *source_key_identifier* parameter.

If internal key tokens are supplied in the *source_key_identifier* or *exporter_key_identifier* parameters, the key in one or both tokens can be reenciphered. This occurs if the master key was changed since the internal key token was last used. The return and reason codes that indicate this do *not* indicate which key was reenciphered. Therefore, assume both keys have been reenciphered.

Key Export (CSNBKEX)

This service cannot be used to export AKEKs. Refer to “ANSI X9.17 Key Export (CSNAKEX)” on page 319 for information on exporting AKEKs.

To use NOCV key-encrypting keys or to export double-length MAC and MACVER keys, the NOCV-enablement keys must be installed in the CKDS.

Key Export operations which specify a NOCV key-encrypting key as the exporter key and also specify a source or key-encrypting key which contains a control vector not supported by the Cryptographic Coprocessor Feature will fail.

If the key type is MACD, the control vectors of the input keys must be the standard control vectors supported by the Cryptographic Coprocessor Feature, since the key export service will be processed on the Cryptographic Coprocessor Feature in this case.

Key Generate (CSNBKGN)

Use the key generate callable service to generate either one or two odd parity DES keys of *any* type.The keys can be single-length (8 bytes), double-length (16 bytes), or, in the case of DATA keys, triple-length (24 bytes). The callable service does not produce keys in clear form and all keys are returned in encrypted form. When two keys are generated, each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature.

ICSF routes the Key Generate request to a PCI Cryptographic Coprocessor if the key type specified in *key_type_1* or *key_type_2* is not valid for the Cryptographic Coprocessor Feature or if the control vector in a supplied token cannot be processed on the Cryptographic Coprocessor Feature. The Key Generate request is also routed to a PCI Cryptographic Coprocessor if a key length of SINGLE-R is specified, or if a key form of OPIM (see note), OPOP or IMIM is specified. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Note: OPIM is valid on the Cryptographic Coprocessor Feature for key forms DATA/DATA, DATAM/DATAM and MAC/MAC. All other OPIM key forms are routed to the PCI Cryptographic Coprocessor.

Format

```
CALL CSNBKGN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_form,
             key_length,
             key_type_1,
             key_type_2,
             kek_key_identifier_1,
             kek_key_identifier_2,
             generated_key_identifier_1,
             generated_key_identifier_2 )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_form

Direction: Input Type: Character string

A 4-byte keyword that defines the type of key(s) you want generated. This parameter also specifies if each key should be returned for either operational, importable, or exportable use. The keyword must be in a 4-byte field, left-justified, and padded with blanks.

The first two characters refer to *key_type_1*. The next two characters refer to *key_type_2*.

The following keywords are allowed: OP, IM, EX, OPIM, OPEX, IMEX, EXEX, OPOP, and IMIM. See Table 12 for their meanings.

Table 12. Key Form Values for the Key Generate Callable Service

Keyword	Meaning
EX	One key that can be sent to another system.
EXEX	A key pair; both keys to be sent elsewhere, possibly for exporting to two different systems. The key pair has the same clear value.
IM	One key that can be locally imported. The key can later be imported onto this system to make it operational.
IMEX	A key pair to be imported; one key to be imported locally and one key to be sent elsewhere. Both keys have the same clear value.

Table 12. Key Form Values for the Key Generate Callable Service (continued)

Keyword	Meaning
IMIM	A key pair to be imported; both keys to be imported locally at a later time.
OP	One operational key. The key is returned to the caller in the key token format. Specify the OP key form when generating AKEKs.
OPEX	A key pair; one key that is operational and one key to be sent from this system. Both keys have the same clear value.
OPIM	A key pair; one key that is operational and one key to be imported to the local system. Both keys have the same clear value. On the other system, the external key token can be imported to make it operational.
OPOP	A key pair; normally with different control vector values.

The key forms are defined as follows:

Operational (OP)
The key value is enciphered under a master key. The result is placed into an internal key token. The key is then operational at the local system. For AKEKs, the result is placed in a skeleton token created by the key token build callable service.

Importable (IM)
The key value is enciphered under an importer key-encrypting key. The result is placed into an external key token.

Exportable (EX)
The key value is enciphered under an exporter key-encrypting key. The result is placed into an external key token. The key can then be transported or exported to another system and imported there for use. This key form cannot be used by any ICSF callable service.

The keys are placed into tokens that the *generated_key_identifier_1* and *generated_key_identifier_2* parameters identify.

If this service is generating only the OP form for a DATA key, it marks the DATA key with the data encryption algorithm according to the system default encryption algorithm unless token copying overrides this. If this service is generating only the OP form for an IMPORTER, it marks the IMPORTER with the SYS-ENC (X'00') data encryption algorithm bits, unless token copying overrides this. Marking of data encryption algorithm bits and token copying are performed only if the service is processed on the Cryptographic Coprocessor Feature. See the *generated_key_identifier_1* parameter for more information on token copying.

Valid key type combinations depend on the key form. See Table 16 for valid key combinations.

key_length

Direction: Input	Type: Character string

An 8-byte value that defines the setting of each part of a double-length or triple-length key. The keyword must be left-justified and padded on the right with blanks. You must supply one of the key length values from Table 13 in the *key_length* parameter.

To generate a single-length key, specify *key_length* as SINGLE or KEYLN8.

Use SINGLE if you want to create a transport key that you would use to exchange DATA keys with a PCF system. Because PCF does not use double-length transport keys, specify SINGLE so that the effects of multiple encipherment are nullified.

The SINGLE-R keyword (″single replicated″) specifies a double-length key where both halves of the key are identical. This key performs as though the key were single length.

Double-length (16-byte) keys have an 8-byte left half and an 8-byte right half. Both halves can have identical clear values or not. If you want the same value to be used in both key halves, specify *key_length* as SINGLE or KEYLN8. If you want different values to be the basis of each key half, specify *key_length* as DOUBLE or KEYLN16.

Triple-length (24-byte) keys have three 8-byte key parts. This key length is valid for DATA keys only. To generate a triple-length DATA key with three different values to be the basis of each key part, specify *key_length* as KEYLN24.

When generating an AKEK, the *key_length* parameter is ignored. The AKEK key length (8-byte or 16-byte) is determined by the skeleton token created by the key token build callable service and provided in the *generated_key_identifier_1* parameter.

The following key types may only be generated through the TOKEN keyword with the appropriate control vector.

Table 13. Key Length Values for the Key Generate Callable Service

Keyword	Meaning
DOUBLE	A 16-byte key with independent 8-byte parts. Valid for CIPHER, CVARDEC, CVARENC, CVARPINE, CVARXCVL, CVARXCVR, DATA, DATAC, DATAM, DATAMV, DECIPHER, DKYGENKY, ENCIPHER, EXPORTER, IKEYXLAT, IMPORTER, IPINENC, KEYGENKY, OKEYXLAT, OPINENC, PINGEN, and PINVER key types. For DATA keys, DOUBLE generates a standard token containing two separate 8-byte DATA keys.
KEYLN8	Synonymous with SINGLE.
KEYLN16	Valid for all key types except DATAXLAT, MAC, and MACVER. Synonymous with DOUBLE.
KEYLN24	Valid for DATA key type only. Generates a standard token containing three separate 8-byte DATA keys.

Table 13. Key Length Values for the Key Generate Callable Service (continued)

Keyword	Meaning
SINGLE	Valid for all key types except DATAC, DATAM, and DATAMV. For 8-byte keys (DATAXLAT, MAC, and MACVER), this is the only valid value for the *key_length* parameter. For DATA keys, SINGLE generates a standard token containing a single-length key. For 16-byte keys (EXPORTER, IMPORTER, IPINENC, OPINENC, PINGEN, PINVER), SINGLE generates effective single-length for those key types. Therefore, each half of the 16-byte key has the same clear value. Note that the following restrictions exist when you specify SINGLE for 16-byte keys. • If you specify *key_length* as SINGLE and key_type_1 or *key_type_2* as any of the double-length keys (EXPORTER, IMPORTER, IPINENC, OPINENC, PINGEN, PINVER), you specify the parameter *key_form* as IMEX, parameter *kek_key_identifier_1 must* contain a NOCV IMPORTER key-encrypting key, either as a key label or an internal key token. • If you specify SINGLE for any double-length keys, the CKDS must contain NOCV enablement keys.
SINGLE-R	A double-length key with equal valued halves.

key_type_1

Direction: Input	Type: Character string

An 8-byte keyword from the list shown in Table 14 on page 87 or the keyword TOKEN. Use the *key_type_1* parameter for the first, or only key, that you want generated. The keyword must be left-justified and padded with blanks. Valid type combinations depend on the key form.

If *key_type_1* is TOKEN, ICSF examines the control vector (CV) field in the *generated_key_identifier_1* parameter to derive the key type. The *generated_key_identifier_1* parameter must be a key token, not a key label. When *key_type_1* is TOKEN, ICSF does not check for the length of the key for DATA keys. Instead, ICSF uses the *key_length* parameter to determine the length of the key.

To generate an AKEK, specify a *key_type_1* of TOKEN. The *generated_key_identifier_1* parameter must be a skeleton token of an AKEK created by the key token build (CSNBKTB) callable service. The token cannot be a partially notarized AKEK or an AKEK key part.

See Table 15 and Table 16 for valid key type and key form combinations.

key_type_2

Direction: Input	Type: Character string

An 8-byte keyword from the list shown in Table 14 on page 87 or the word TOKEN. Use the *key_type_2* parameter for a key pair, which is shown in

Table 16 on page 91. The keyword must be left-justified and padded with blanks. Valid type combinations depend on the key form.

If *key_type_2* is TOKEN, ICSF examines the control vector (CV) field in the *generated_key_identifier_2* parameter to derive the key type. When *key_type_2* is TOKEN, ICSF does not check for the length of the key for DATA keys. Instead, ICSF uses the *key_length* parameter to determine the length of the key.

Note: Key types marked with an ″*″ must be requested through the specification of a proper control vector in a key token and through the use of the TOKEN keyword.

If you want only one key to be generated, specify the *key_type_2* and *KEK_key_identifier_2* as binary zeros.

Table 14. Key Type Values for the Key Generate Callable Service

Key Type	Meaning
CIPHER	Used only to encrypt or decrypt data. Possible key lengths are single-length or double-length. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
CVARDEC*	The TSS Cryptographic variable decipher verb uses a CVARDEC key to decrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.
CVARENC*	Cryptographic variable encipher service uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.
CVARPINE*	Used to encrypt a PIN value for decryption in a PIN-printing application. Possible key lengths are single-length or double-length.
CVARXCVL*	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
CVARXCVR*	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
DATA	Data-encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAC*	Used to specify a DATA-class key that will perform in the Encipher and Decipher callable services, but not in the MAC Generate or MAC Verify callable services. This is a double-length key.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.

Table 14. Key Type Values for the Key Generate Callable Service (continued)

Key Type	Meaning
DECIPHER	Used only to decrypt data. Possible key lengths are single-length or double-length. DECIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
DKYGENKY*	Used to generate a diversified key based on the key-generating key. This is a double-length key.
ENCIPHER	Used only to encrypt data. Possible key lengths are single-length or double-length. ENCIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
EXPORTER	Exporter key-encrypting key. Use this double-length key to convert any key (including a DATA key) from operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IPINENC	Input PIN-encrypting key. Use this double-length input key to translate PINs. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate and Encrypted PIN verify callable services.
KEYGENKY*	Used to generate a key based on the key-generating key. This is a double-length key.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Double-length output PIN-encrypting key. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.

See Table 15 on page 91 and Table 16 on page 91 for valid key type and key form combinations.

KEK_key_identifier_1

Direction: Input/Output   Type: String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label. If you supply a key label that is less than 64-bytes, it must be left-justified and padded with blanks. *KEK_key_identifier_1* is required for a *key_form* of IM, EX, IMEX, EXEX, or IMIM.

If the *key_form* is OP, OPEX, OPIM, or OPOP, the *KEK_key_identifier_1* is null.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key. For example, the key has been installed in the cryptographic key data set through the key generator utility program or the key entry hardware using the NOCV parameter; or you are passing the key-encrypting key in the internal key token with the NOCV bit on and your program is running in supervisor state or key 0-7.

Control vectors are explained in "Control Vector" on page 16 and the NOCV bit is shown in Table 97 on page 365.

KEK_key_identifier_2

Direction: Input/Output	Type: String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label of an internal token. If you supply a key label that is less than 64-bytes, it must be left-justified and padded with blanks. *KEK_key_identifier_2* is required for a *key_form* of OPIM, OPEX, IMEX, IMIM, or EXEX. This field is ignored for *key_form* keywords OP, IM and EX.

For DATA and KEK (importer and exporter) types, the key generate service propagates the data encryption algorithm bits from the KEK you supply to the key it generates, unless overridden by token copying. In generating an IMPORTER or EXPORTER OP KEK, this service marks the key CDMF, DES, or SYS-ENC, corresponding to *kek_key_identifier_2*. In generating an OP DATA key, if the *kek_key_identifier_2* is SYS-ENC, this service marks the key according to the system default encryption algorithm. Otherwise, it marks the key CDMF or DES, corresponding to the *kek_key_identifier_2*. Propagation of token markings is only relevant when this service is processed on the Cryptographic Coprocessor Feature. For more information on token copying, see the *generated_key_identifier_1* parameter.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key. For example, the key has been installed in the cryptographic key data set through the key generator utility program or the key entry hardware using the NOCV parameter; or you are passing the key-encrypting key in the internal key token with the NOCV bit on and your program is running in supervisor state or in key 0-7.

Control vectors are explained in "Control Vector" on page 16 and the NOCV bit is shown in Table 97 on page 365.

generated_key_identifier_1

Direction: Input/Output	Type: String

This parameter specifies either a generated:

- Internal key token for an operational key form, or
- External key token containing a key enciphered under the *kek_key_identifier_1* parameter.

If you specify a *key_type_1* of TOKEN, then this field contains a valid token of the key type you want to generate. See Table 14 on page 87 for a list of valid *key_type_1* key types.

If you specify a *key_type_1* of IMPORTER or EXPORTER and a *key_form* of OPEX, and if the *generated_key_identifier_1* parameter contains a valid internal token of the SAME type, the NOCV bit, if on, is propagated to the generated key token.

Note: Propagation of the NOCV bit is performed only if the service is processed on the Cryptographic Coprocessor Feature.

If you specify a *key_type_1* of DATA or IMPORTER, and a *key_form* of OP, and the *generated_key_identifier_1* parameter contains a valid internal token of the same type, the data encryption algorithm bits are copied to the generated key token. This overrides any markings on the *kek_key_identifier_2*. If you specify a *key_type_1* of DATA, IMPORTER, or EXPORTER and a *key_form* of OPIM or OPEX and the *generated_key_identifier_1* parameter contains a valid internal token of the same type, the data encryption algorithm bits are copied to the generated key token. This overrides any data encryption algorithm bit markings on the *kek_key_identifier_2*.

Note: Marking of data encryption algorithm bits and token copying are performed only if the service is processed on the Cryptographic Coprocessor Feature.

When generating an AKEK, specify the skeleton key token created by the key token build callable service (CSNBKTB) as input for this parameter.

generated_key_identifier_2

Direction: Input/Output Type: String

This parameter specifies a generated external key token containing a key enciphered under the *kek_key_identifier_2* parameter.

If you specify a *key_type_2* of TOKEN, then this field contains a valid token of the key type you want to generate. See Table 14 on page 87 for a list of the valid *key_type_2* key types. The token can be an internal or external token.

Restriction

The caller must be in task mode, not in SRB mode.

Usage Notes

No external tokens that this service generates contain data encryption algorithm bit markings.

It is possible to generate an operational DES-marked DATA key on a CDMF-only system or a CDMF-marked DATA key on a DES-only system. However, the encipher (CSNBENC) and decipher (CSNBDEC) callable services fail when you use these keys on the systems where they were generated unless overridden by keyword.

Double-length MAC keys (DATAM and DATAMV) can be generated in the same key forms as single-length MAC keys.

Table 15 shows the valid key type and key form combinations for a single key. Key types marked with an ″*″ must be requested through the specification of a proper control vector in a key token and through the use of the TOKEN keyword.

Table 15. Key Generate Valid Key Types and Key Forms for a Single Key

Key Type 1	Key Type 2	OP	IM	EX
DATA	Not applicable	X	X	X
DATAC*	Not applicable	X	X	X
DATAM	Not applicable	X	X	X
DKYGENKY*	Not applicable	X	X	X
KEYGENKY*	Not applicable	X	X	X
MAC	Not applicable	X	X	X
PINGEN	Not applicable	X	X	X

Table 16 shows the valid key type and key form combinations for a key pair. Key types marked with an ″*″ must be requested through the specification of a proper control vector in a key token and through the use of the TOKEN keyword.

Table 16. Key Generate Valid Key Types and Key Forms for a Key Pair

Key Type 1	Key Type 2	OPEX	EXEX	OPIM, OPOP, IMIM	IMEX
CIPHER	CIPHER	X	X	X	X
CIPHER	DECIPHER	X	X	X	X
CIPHER	ENCIPHER	X	X	X	X
CVARDEC*	CVARENC*	X			X
CVARDEC*	CVARPINE*	X			X
CVARENC*	CVARDEC*	X			X
CVARENC*	CVARXCVL*	X			X
CVARENC*	CVARXCVR*	X			X
CVARXCVL*	CVARENC*	X			X
CVARXCVR*	CVARENC*	X			X
CVARPINE*	CVARDEC*	X			X
DATA	DATA	X	X	X	X
DATA	DATAXLAT	X	X		X
DATAC*	DATAC*	X	X	X	X
DATAM	DATAM	X	X	X	X
DATAM	DATAMV	X	X	X	X
DATAXLAT	DATAXLAT	X	X		X
DECIPHER	CIPHER	X	X	X	X

Table 16. Key Generate Valid Key Types and Key Forms for a Key Pair (continued)

Key Type 1	Key Type 2	OPEX	EXEX	OPIM, OPOP, IMIM	IMEX
DECIPHER	ENCIPHER	X	X	X	X
DKYGENKY*	DKYGENKY*	X	X	X	X
ENCIPHER	CIPHER	X	X	X	X
ENCIPHER	DECIPHER	X	X	X	X
EXPORTER	IKEYXLAT	X	X		X
EXPORTER	IMPORTER	X	X		X
IKEYXLAT	EXPORTER	X	X		X
IKEYXLAT	OKEYXLAT	X	X		X
IMPORTER	EXPORTER	X	X		X
IMPORTER	OKEYXLAT	X	X		X
IPINENC	OPINENC	X	X	X	X
KEYGENKY*	KEYGENKY*	X	X	X	X
MAC	MAC	X	X	X	X
MAC	MACVER	X	X	X	X
OKEYXLAT	IKEYXLAT	X	X		X
OKEYXLAT	IMPORTER	X	X		X
OPINENC	IPINENC	X	X	X	X
OPINENC	OPINENC			X	
PINVER	PINGEN	X	X		X
PINGEN	PINVER	X	X		X

Note: If the *key_form* is IMEX, the *key_length* is SINGLE, and *key_type_1* is IPINENC, OPINENC, PINGEN, IMPORTER, or EXPORTER, you must specify the *kek_key_identifier_1* parameter as NOCV IMPORTER.

To use NOCV key-encrypting keys, NOCV-enablement keys must be installed in the CKDS.

To generate DATAM and DATAMV keys in the importable form, the ANSI system keys must be installed in the CKDS.

Key Import (CSNBKIM)

Use the key import callable service to reencipher a key (except an AKEK) from encryption under an importer key-encrypting key to encryption under the master key. The reenciphered key is in operational form.

This service examines the data encryption algorithm bits on the operational KEK supplied. It propagates the DES, CDMF, or SYS-ENC markings on the KEK token to the imported KEK or DATA key token unless token copying overrides this. It propagates the SYS-ENC marking to the KEK and marks the DATA key according to the system's default encryption algorithm. Propagation of token markings is only relevant when this service is processed on the Cryptographic Coprocessor Feature. See the *target_key_identifier* and *importer_key_identifier* parameters for more information.

For DATA keys, this service generates a key of the same length as that contained in the input token.

ICSF routes the Key Import request to a PCI Cryptographic Coprocessor if the control vector in a supplied internal token cannot be processed on the Cryptographic Coprocessor Feature, or if the key type is not valid for the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Format

```
CALL CSNBKIM(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_type,
             source_key_identifier,
             importer_key_identifier,
             target_key_identifier )
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

key_type

Direction: Input Type: Character string

The type of key you want to reencipher under the master key. Specify an 8-byte keyword or the keyword TOKEN. The keyword must be left-justified and padded on the right with blanks. The key types you can specify are listed in Table 17.

To import a double-length MAC generation key, use a key type of DATAM or TOKEN. To import a double-length MAC verification key, use a key type of DATAMV or TOKEN.

For a double-length MAC key with a key type of DATAM, the source key identifier must specify an external token that uses the data compatibility control vector. For a double-length MAC key with a key type of MACD, the source key identifier must specify an external token that uses the single-length control vector for both the left and right half of the key (MAC ∥ MAC). For a table of control vectors, refer to “Control Vector Table” on page 383.

If the key type is TOKEN, ICSF determines the key type from the control vector (CV) field in the external key token provided in the *source_key_identifier* parameter. If the control vector is invalid on the Cryptographic Coprocessor Feature, the key import request will be routed to the PCI Cryptographic Coprocessor.

Table 17. Key Type Values for the Key Import Callable Service

Key Type	Meaning
AKEK	ANSI X9.17 key-encrypting key. A single-length or double-length key that must be ANSI notarized and offset before use as a key-encrypting key. The default is double-length.
CIPHER	Used only to encrypt or decrypt data. Possible key lengths are single-length or double-length. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
DATA	Data-encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAC	Used to specify a DATA-class key that will perform in the Encipher and Decipher callable services, but not in the MAC Generate or MAC Verify callable services. This is a double-length key.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
DECIPHER	Used only to decrypt data. Possible key lengths are single-length or double-length. DECIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.

Table 17. Key Type Values for the Key Import Callable Service (continued)

Key Type	Meaning
ENCIPHER	Used only to encrypt data. Possible key lengths are single-length or double-length. ENCIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
EXPORTER	Exporter key-encrypting key. Use this double-length key to convert any key (including a DATA key) from operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IMP-PKA	Limited authority importer key. Use this double-length key to import an encrypted RSA or DSS private key.
IPINENC	Input PIN-encrypting key. Use this double-length input key to translate PINs. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate and Encrypted PIN verify callable services.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACD	Double-length MAC generation and verification key. ICSF continues to support this key type for compatibility with ICSF/MVS Version 2 Release 1.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Double-length output PIN-encrypting key. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.

source_key_identifier

Direction: Input   Type: String

The key you want to reencipher under the master key. The parameter is a 64-byte field for the enciphered key to be imported containing either an external key token or a null key token. If you specify a null token, the token is all binary zeros, except for a key in bytes 16-23 or 16-31, or in bytes 16-31 and 48-55 for triple-length DATA keys. Refer to Table 99 on page 368.

If key type is TOKEN, this field may not specify a null token.

This service supports the no-export function in the CV.

importer_key_identifier

Direction: Input/Output	Type: String

The importer key-encrypting key that the key is currently encrypted under. The parameter is a 64-byte area containing either the key label of the key in the cryptographic key data set or the internal key token for the key. If you supply a key label that is less than 64-bytes, it must be left-justified and padded with blanks.

This service examines data encryption algorithm bits on the operational KEK token supplied. It propagates DES or CDMF markings on the KEK token to the imported KEK or DATA key token unless token copying overrides this. For SYS-ENC marking, this service propagates the marking to the KEK or marks the DATA key according to the default system encryption algorithm, unless token copying overrides this.

Note: If you specify a NOCV importer in the *importer_key_identifier* parameter, the key to be imported must be enciphered under the importer key itself.

target_key_identifier

Direction: Input/Output	Type: String

This parameter is the generated reenciphered key. The parameter is a 64-byte area that receives the internal key token for the imported key.

If the imported key TYPE is IMPORTER or EXPORTER and the token key TYPE is the same, the *target_key_identifier* parameter changes direction to both input and output. If the application passes a valid internal key token for an IMPORTER or EXPORTER key in this parameter, the NOCV bit is propagated to the imported key token.

Note: Propagation of the NOCV bit is performed only if the service is processed on Cryptographic Coprocessor Feature.

If the *key_type* is DATA, IMPORTER, or EXPORTER and the application passes a valid internal key token for a DATA, IMPORTER, or EXPORTER key in the *target_key_identifier,* this service propagates the data encryption algorithm bits to the imported key token. Any data encryption algorithm bits supplied on *importer_key_identifier* are ignored.

Note: Marking of data encryption algorithm bits and token copying are performed only if the service is processed on the Cryptographic Coprocessor Feature.

Usage Notes

For the key import callable service, choose one of the following options:

- Specify the *key_type* parameter as TOKEN and specify the external key token in the *source_key_identifier* parameter. The key type information is determined from the control vector in the external key token.

- Specify a key type in the *key_type* parameter and specify an external key token in the *source_key_identifier* parameter. The specified key type must be compatible with the control vector in the external key token.
- Specify a valid key type in the *key_type* parameter and a null key token in the *source_key_identifier* parameter. Use the control vector that maps to the specification of the *key_type* parameter.

The key import callable service cannot be used to import ANSI key-encrypting keys. For information on importing these types of keys, refer to "ANSI X9.17 Key Import (CSNAKIM)" on page 324.

If the key type is MACD or IMP-PKA, the control vectors of supplied internal tokens must all be supported by the Cryptographic Coprocessor Feature, since processing for these key types will not be routed to a PCI Cryptographic Coprocessor.

For Source key encrypted without CVs, the length of DATA keys is determined by non-zero enciphered key values.

To use NOCV key-encrypting keys or to import DATAM or DATAMV keys, NOCV-enablement keys must be installed in the CKDS.

Key Import operations which specify a NOCV key-encrypting key as either the importer key or the target and also specify a source or key-encrypting key which contains a control vector not supported by the Cryptographic Coprocessor Feature will fail.

Note: To import a double-length MAC generation key encrypted under a NOCV KEK, use the DATAM key type value. To import a double-length MAC verification key encrypted under a NOCV KEK, use the DATAMV key type value. In either case, do not specify a key type value of TOKEN.

Key Part Import (CSNBKPI)

Use the key part import callable service to combine the clear key parts of any key type and return the combined key value either in an internal token or as an update to the CKDS.

Before you use the key part import service for the first key part, you must use the key token build service to create the internal key token into which the key will be imported. Subsequent key parts are combined with the first part in internal token form or as a label from the CKDS.

Key parts are specified as FIRST, MIDDLE, or LAST in the *rule_array*. Only when the LAST part has been combined can the key token be used in any other service.

The key part import callable service can also be used to import a key without using key parts. Call the key part import service FIRST with key part value X'0000...' then call the key part import service LAST with the complete value.

Keys created via this service have odd parity. The FIRST key part is adjusted to odd parity. All subsequent key parts are adjusted to even parity before being combined.

ICSF routes all requests to the PCI Cryptographic Coprocessor except for key type of AKEK. AKEK is always processed on the Cryptographic Coprocessor Feature.

Format

```
CALL CSNBKPI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_part,
            key_identifier)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input	Type: String

The keyword that provides control information to the callable service. The keywords must be 8 bytes of contiguous storage with the keyword left-justified in its 8-byte location and padded on the right with blanks.

Table 18. Keywords for Key Part Import Control Information

Keyword	Meaning
Key Part (Required)	
FIRST	This keyword specifies that an initial key part is being entered.
LAST	This keyword specifies that the last key part is being entered.
MIDDLE	This keyword specifies that an intermediate key part, which is neither the first key part nor the last key part, is being entered.

key_part

Direction: Input	Type: String

A 16-byte field containing the clear key part to be entered. If the key is a single-length key, the key part must be left-justified and padded on the right with zeros.

key_identifier

Direction: Input/Output	Type: String

A 64-byte field containing an internal token or a label of an existing CKDS record. If *rule_array* is FIRST, this field is the skeleton of an internal token of a single- or double-length key with the KEY-PART marking. If *rule_array* is MIDDLE or LAST, this is an internal token or the label of a CKDS record of a partially combined key. Depending on the input format, the accumulated partial or complete key is returned as an internal token or as an updated CKDS record. The returned *key_identifier* will be encrypted under the current master key.

Restriction

The caller must be in task mode and must not be in cross memory mode. If a label is specified on *key_identifier*, the label must be unique. If more than one record is found, the service fails.

Usage Note

This service requires that the ANSI system keys be installed on the CKDS.

Related Information

This service is consistent with the Transaction Security System key part import verb.

Key Record Create (CSNBKRC)

Use the key record create callable service to add a key record to the CKDS. The record contains a key token set to binary zeros and is identified by the label passed in the *key_label* parameter. This service updates both the DASD copy of the CKDS currently in use by ICSF and the in-storage copy of the CKDS.

Format

```
CALL CSNBKRC(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_label)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

key_label

Direction: Input   Type: Character string

The 64-byte label of a record in the CKDS that is the target of this service. The created record contains a key token set to binary zeros and has a key type of NULL.

Restrictions

The caller must be in task mode and must not be in cross memory mode. The record must have a unique label. Therefore, there cannot be another record in the CKDS with the same label and a different key type.

Usage Notes

The key record create callable service checks the syntax of the label provided in the *key_label* parameter to ensure that it follows the KGUP rules. To bypass label syntax checking, use a preprocessing exit to turn on the bypass parse bit in the Exit Parameter Control Block (EXPB). For more information about preprocessing exits and the EXPB, refer to the *z/OS ICSF System Programmer's Guide*.

You must use either the key record create callable service or KGUP to create an initial record in the CKDS before you can use the key record write service to update the record with a valid key token. Your applications perform better if you use KGUP to create the initial records and REFRESH the entire in-storage copy of the CKDS, rather than using key record create to create the initial NULL key entries. This is particularly true if you are creating a large number of key records. Key record create adds a record to a portion of the CKDS that is searched sequentially during key retrieval. Using KGUP followed by a REFRESH puts the null key records in the portion of the CKDS that is ordered in key-label/type sequence. A binary search of the key-label/type sequenced part of the CKDS is more efficient than searching the sequentially ordered section.

Key Record Delete (CSNBKRD)

Use the key record delete callable service to delete a key record from both the DASD copy of the CKDS and the in-storage copy.

Format

```
CALL CSNBKRD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_label)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords supplied in the rule_array parameter. This number must always be 1.

rule_array

Direction: Input Type: Character string

The 8 byte keyword that defines the action to be performed. The keyword must be LABEL-DL.

key_label

Direction: Input Type: Character string

The 64-byte label of a record in the CKDS that is the target of this service. The record pointed to by this label is deleted.

Restrictions

The caller must be in task mode and must not be in cross memory mode. The record defined by the *key_label* must be unique. If more than one record per label is found, the service fails.

Key Record Read (CSNBKRR)

Use the key record read callable service to copy an internal key token from the in-storage CKDS to application storage. Other cryptographic services can then use the copied key token directly. The key token can also be used as input to the token copying functions of key generate, key import, or secure key import services to create additional NOCV keys.

Format

```
CALL CSNBKRR(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_label,
            key_token)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it indicating specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

key_label

Direction: Input   Type: Character string

The 64-byte label of a record in the in-storage CKDS. The internal key token in this record is returned to the caller.

key_token

Direction: Output   Type: String

The 64-byte internal key token retrieved from the in-storage CKDS.

Restrictions

The record defined by the *key_label* parameter must be unique and must already exist in the CKDS.

Key Record Write (CSNBKRW)

Use the key record write callable service to write an internal key token to the CKDS record specified by the *key_label* parameter. This service supports writing a record to the CKDS which contains a key token with a control vector which is not supported by the Cryptographic Coprocessor Feature. These records will be written to the CKDS with a key type of CV, unless the key is an IMPORTER, EXPORTER, PINGEN, PINVER, IPINENC, or OPINENC type. These key types will be preserved in the CKDS record, even if the control vector is not supported by the Cryptographic Coprocessor Feature. This service updates both the DASD copy of the CKDS currently in use by ICSF and the in-storage copy. The record you are updating must be unique and must already exist in both the DASD and in-storage copies of the CKDS.

Format

```
CALL CSNBKRW(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_token,
            key_label)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_token

Direction: Input/output Type: String

The 64-byte internal key token that is written to the CKDS.

key_label

Direction: Input Type: Character string

The 64-byte label of a record in the CKDS that is the target of this service. The record is updated with the internal key token supplied in the *key_token* parameter.

Restrictions

The caller must be in task mode and must not be in cross memory mode. The record defined by the *key_label* parameter must be unique and must already exist in the CKDS.

Related Information

You can use this service with the key record create callable service to write an initial record to key storage. Use it following the key import and key generate callable services to write an operational key imported or generated by these services directly to the CKDS.

Key Test and Key Test Extended (CSNBKYT and CSNBKYTX)

Use the key test callable service to generate or verify a secure, cryptographic verification pattern for keys. The key to test can be in the clear or encrypted under the master key. The key test extended callable service also supports keys encrypted under a key-encrypting key (KEK). Keywords in the rule array specify whether the callable service generates or verifies a verification pattern.

When the service generates a verification pattern, it creates and cryptographically processes a random number. The service returns the random number with the verification pattern.

When the service tests a verification pattern against a key, you must supply the random number and the verification pattern from a previous call to key test or key test extended. The service returns the verification result in the return and reason codes.

CSNBKYT is consistent with the Transaction Security System verb of the same name. If you generate a key on the Transaction Security System, you can verify it on ICSF and vice versa.

Key Test and Key Test Extended (CSNBKYT and CSNBKYTX)

The key test callable service does not support triple-length DATA keys. CSNBKYT will be routed to a PCI Cryptographic Coprocessor for processing if ANSI enablement keys are not installed in the CKDS. ENC-ZERO requests are routed to the PCI Cryptographic Coprocessor.

The CSNBKYTX callable service is processed on the Cryptographic Coprocessor Feature. *Rule_array* keywords KEY-CLR, KEY-CLRD, and ENC-ZERO are not valid for CSNBKYTX.

Format

```
CALL CSNBKYT(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_identifier,
            random_number,
            verification_pattern)
```

```
CALL CSNBKYTX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_identifier,
             random_number,
             verification_pattern,
             kek_key_identifier)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value can be 2 or 3.

rule_array

Direction: Input	Type: String

Two or three keywords that provide control information to the callable service. Table 19 lists the keywords. The keywords must be in 16 or 24 bytes of contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks.

Table 19. Keywords for Key Test and Key Test Extended Control Information

Keyword	Meaning
Key Rule (required)	
KEY-CLR	Specifies the key supplied in *key_identifier* is a single-length clear key. This keyword is not valid for the key test extended callable service.
KEY-CLRD	Specifies the key supplied in *key_identifier* is a double-length clear key. This keyword is not valid for the key test extended callable service.
KEY-ENC	Specifies the key supplied in *key_identifier* is a single-length encrypted key.
KEY-ENCD	Specifies the key supplied in *key_identifier* is a double-length encrypted key.
Process Rule (required)	
GENERATE	Generate a verification pattern for the key supplied in *key_identifier*.
VERIFY	Verify a verification pattern for the key supplied in *key_identifier*.
Parity Adjustment (optional)	
ADJUST	Adjust the parity of test key to odd before generating or verifying the verification pattern. The *key_identifier* field itself is not adjusted.
NOADJUST	Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.
Verification Process Rule (optional)	

Table 19. Keywords for Key Test and Key Test Extended Control Information (continued)

Keyword	Meaning
ENC-ZERO	Specifies use of the ″encrypted zeros″ method. Use only with KEY-CLR, KEY-CLRD, KEY-ENC, or KEY-ENCD keywords. This keyword is not valid for the key test extended callable service.

key_identifier

Direction: Input/Output Type: String

The key for which to generate or verify the verification pattern. The parameter is a 64-byte string of an internal token, key label, or a clear key value left-justified. In the CSNBKYTX service, this parameter can also be an external token.

Note: If you supply a key label for this parameter, it must be unique on the CKDS.

random_number

Direction: Input/Output Type: String

This is an 8-byte field that contains a random number supplied as input for the test pattern verification process and returned as output with the test pattern generation process.

verification_pattern

Direction: Input/Output Type: String

This is an 8-byte field that contains a verification pattern supplied as input for the test pattern verification process and returned as output with the test pattern generation process.

kek_key_identifier

Direction: Input/Output Type: String

This parameter is for the CSNBKYTX service only. If *key_identifier* is an external token, then this is a 64-byte string of an internal token or a key label of an IMPORTER or EXPORTER used to encrypt the test key.

Note: If you supply a key label for this parameter, it must be unique on the CKDS.

Usage Notes

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms, that is, clear, operational or external.

In the Transaction Security System, KEY-ENC and KEY-ENCD both support enciphered single-length and double-length keys. They use the key-form bits in byte 5 of CV to determine the length of the key. To be consistent, in ICSF, both KEY-ENC and KEY-ENCD handle single- and double-length keys. Both products effectively ignore the keywords, which are supplied only for compatibility reasons.

Rule_array keywords KEY-CLR, KEY-CLRD, and ENC-ZERO are not valid for the CSNBKYTX callable service.

Key Token Build (CSNBKTB)

Use the key token build callable service to build an external or internal key token from information which you supply. The token can be used as input for the key generate and key part import callable services. You can specify a control vector or the service can build a control vector based upon the key type you specify and the control vector-related keywords in the rule array. ICSF supports the building of an internal key token with the key encrypted under a master key other than the current master key.

You can also use this service to update the DES, CDMF, or SYS-ENC markings in a supplied DATA, IMPORTER, or EXPORTER token and to build CCA key tokens for all key types ICSF supports.

Format

```
CALL CSNBKTB(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_token,
            key_type,
            rule_array_count,
            rule_array,
            key_value,
            master_key_version_number,
            key_register_number,
            secure_token,
            control_vector,
            initialization_vector,
            pad_character,
            cryptographic_period_start,
            masterkey_verify_parm
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

Reserved field.

exit_data

Direction: Input/Output Type: String

Reserved field.

key_token

Direction: Input/Output Type: String

If the following parameter *key_type* is TOKEN then this is a 64-byte internal token that is updated as specified in the *rule_array*. The internal token must be a DATA, IMPORTER or EXPORTER key type. Otherwise this field is an output-only field.

key_type

Direction: Input Type: String

An 8-byte field that specifies the type of key you want to build or the keyword TOKEN for updating a supplied token. If *key_type* is TOKEN, then the *key_token* field cannot contain a double- or triple-length DATA key token. For a list of keywords, see Table 20. No other keywords are valid. The TOKEN keyword indicates changing an internal token in the *key_token* parameter. A valid *key_type* indicates building a key token from the parameters specified.

Table 20. Key Type Values for the Key Token Build Callable Service

Key Type	Meaning
AKEK	ANSI X9.17 key-encrypting key. A single-length or double-length key that must be ANSI notarized and offset before use as a key-encrypting key. The default is double-length.
CIPHER	Used only to encrypt or decrypt data. Possible key lengths are single-length or double-length. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
CVARDEC	The TSS Cryptographic variable decipher verb uses a CVARDEC key to decrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.
CVARENC	Cryptographic variable encipher service uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. Possible key lengths are single-length or double-length.

Table 20. Key Type Values for the Key Token Build Callable Service (continued)

Key Type	Meaning
CVARPINE	Used to encrypt a PIN value for decryption in a PIN-printing application. Possible key lengths are single-length or double-length.
CVARXCVL	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
CVARXCVR	Used to encrypt special control values in DES key management. Possible key lengths are single-length or double-length.
DATA	Data-encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAC	Used to specify a DATA-class key that will perform in the Encipher and Decipher callable services, but not in the MAC Generate or MAC Verify callable services. This is a double-length key.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
DECIPHER	Used only to decrypt data. Possible key lengths are single-length or double-length. DECIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
DKYGENKY	Used to generate a diversified key based on the key-generating key.
ENCIPHER	Used only to encrypt data. Possible key lengths are single-length or double-length. ENCIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) callable services.
EXPORTER	Exporter key-encrypting key. Use this double-length key to convert any key (including a DATA key) from operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IPINENC	Input PIN-encrypting key. Use this double-length input key to translate PINs. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate and Encrypted PIN verify callable services.
KEYGENKY	Used to generate a key based on the key-generating key. This is a double-length key.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.

Table 20. Key Type Values for the Key Token Build Callable Service (continued)

Key Type	Meaning
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Double-length output PIN-encrypting key. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.
SECMSG	Used to encrypt PINs or keys in a secure message. This is a double-length key.
USE-CV	Specifies that the key type should be obtained from the control vector specified in the *control_vector* parameter. The CV rule array keyword should be specified if USE-CV is specified.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you supplied in the *rule_array* parameter.

rule_array

Direction: Input   Type: String

One to four keywords that provide control information to the callable service. See Table 21 for a list. The keywords must be in 8 to 32 bytes of contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. For any key type, there are no more than four valid *rule_array* values.

If you specify TOKEN for the *key_type*, then the only valid *rule_array* values are INTERNAL and DES, CDMF, or SYS-ENC. The Data Encryption Algorithm (see the table that follows) keyword has no default. If you specify a *key_type* of DATA, IMPORTER or EXPORTER, the Data Encryption Algorithm selection keyword defaults to SYS-ENC. The other *rule_array* keywords do not apply.

Table 21. Keywords for Key Token Build Control Information

Keyword	Meaning
Token Type (required)	
EXTERNAL	Specifies an external key token.
INTERNAL	Specifies an internal key token.
Key Status (optional)	
KEY	This keyword indicates that the key token to build will contain an encrypted key. The *key_value* parameter identifies the field that contains the key.

Table 21. Keywords for Key Token Build Control Information (continued)

Keyword	Meaning
NO-KEY	This keyword indicates that the key token to build will not contain a key. This is the default key status.
Data Encryption Algorithm (optional) — valid only for single-length DATA keys and KEKs.	
CDMF	For a DATA key, this keyword indicates marking the key token as usable only with the CDMF. For a KEK, it indicates marking the key token to specify that the KEK can encrypt only a CDMF DATA key.
DES	For a DATA key, this keyword indicates marking the key token as usable only with the DES. For a KEK, it indicates marking the key token to specify that the KEK can encrypt only a DES DATA key.
SYS-ENC	For a KEK, this keyword indicates marking the key token as B'00'. This means that the KEK can protect CDMF- or DES-marked DATA key tokens. For a DATA key, this indicates marking the key token according to the encryption algorithm of the system. For a CDMF system, the DATA key token is marked CDMF; for a DES-CDMF or DES system, it is marked DES. When *key_type* of DATA, IMPORTER or EXPORTER is specified, SYS-ENC is the default.
CV on the Link Specification (optional) — valid only for IMPORTER and EXPORTER.	
CV-KEK	This keyword indicates marking the KEK as a CV KEK. The control vector is applied to the KEK before use in encrypting other keys. This is the default.
NOCV-KEK	This keyword indicates marking the KEK as a NOCV KEK. The control vector is not applied to the KEK before use in encrypting other keys. Services using NO-CV keys must be processed on the Cryptographic Coprocessor Feature.
CV Status optional)	
CV	This keyword indicates to obtain the control vector from the variable identified by the *control_vector* parameter.
NO-CV	Default. This keyword indicates that the control vector is to be supplied based on the key type and the control vector related keywords.
Key Length Keywords (optional)	
DOUBLE	Double-length or 16-byte key. Synonymous with KEYLN16. **Note:** See Table 22 on page 116 for valid key types for these key length values.
DOUBLE-O	Double-length 16-byte key valid only for AKEK. Both halves may have the same clear values. DOUBLE-O is mutually exclusive with SINGLE keyword. For AKEKs, DOUBLE-O is the default. DOUBLE-O AKEKs generated by CSNBKGN will have different left and right clear values.
KEYLN8	Single-length or 8-byte key.
KEYLN16	Double-length or 16-byte key.
KEYLN24	Triple-length, 24-byte key valid only for a DATA key type.
SINGLE	Single-length or 8-byte key. Synonymous with KEYLN8.
Key Part Indicator (optional)	

Table 21. Keywords for Key Token Build Control Information (continued)

Keyword	Meaning
KEY-PART	This token is to be used as input to the key part import service.
Control Vector Keywords. Specify one or more of the following (optional)	
See Table 22 on page 116 for the key-usage keywords that can be specified for a given key type.	
Master Key Verification Pattern (optional)	
MKVP	This keyword indicates that the *key_value* is enciphered under the master key which corresponds to the master key verification pattern specified in the *masterkey_verify_parm* parameter. If this keyword is not specified, the key contained in the *key_value* field must be enciphered under the current master key.

key_value

Direction: Input   Type: String

If you use the KEY keyword, this parameter is a 16-byte string that contains the encrypted key value. Single-length keys must be left-justified in the field and padded on the right with X'00'. If you are building a triple-length DATA key, this parameter is a 24-byte string containing the encrypted key value. If you supply an encrypted key value and also specify INTERNAL, the service will check for the presence of the MKVP keyword. If MKVP is present, the service will assume the *key_value* is enciphered under the master key which corresponds to the master key verification pattern specified in the *masterkey_verify_parm* parameter, and will place the key into the internal token along with the verification pattern from the *masterkey_verify_parm* parameter. If MKVP is not specified, ICSF assumes the key is enciphered under the current host master key and places the key into an internal token along with the verification pattern for the current master key. In this case, the application must ensure that the master key has not changed since the key was generated or imported to this system. Otherwise, use of this parameter is not recommended.

master_key_version_number

Direction: Input   Type: Integer

This field is examined only if the KEY keyword is specified, in which case, this field must be zero. If the KEY and INTERNAL keywords are specified in *rule_array*, the service will check for the existence of the MKVP rule array keyword. If MKVP is specified, the service will make use of the last parameter specified (*masterkey_verify_parm*). The service assumes the key provided by the *key_value* parameter is enciphered under the corresponding master key and will place the key into the internal token along with the verification pattern from the *masterkey_verify_parm* parameter.

key_register_number

Direction: Input   Type: Integer

This field is ignored.

secure_token

Direction: Input	Type: String

This field is ignored.

control_vector

Direction: Input	Type: String

A pointer to a 16 byte string variable. If this parameter is specified, and you use the CV rule array keyword, the variable is copied to the control vector field of the key token. See "Control Vector Table" on page 383 for additional information.

initialization_vector

Direction: Input	Type: String

This field is ignored.

pad_character

Direction: Input	Type: Integer

The only allowed value for key types MAC and MACVER is 0. This field is ignored for all other key types.

cryptographic_period_start

Direction: Input	Type: String

This field is ignored.

masterkey_verify_parm

Direction: Input	Type: String

A pointer to an 8-byte string variable. The value is inserted into the key token when you specify both the KEY and INTERNAL keywords in rule array.

Usage Notes

You can use this service to create skeleton key tokens with the desired data encryption algorithm bits for use in some key management services to override the default system specifications.

- To generate an operational CDMF DATA key, build an internal DATA key token with the CDMF keyword and pass that token to the key generate service in the *generated_key_identifier_1* parameter. This generates the desired DATA key independently of the system encryption algorithm default. Similarly, for token copying to override the system default data encryption algorithm bits, you can use this service to build a skeleton token for input to the key generate, key import, or secure key import callable services.

- To generate operational AKEKs, use *key_type* of TOKEN and provide a skeleton AKEK key token as the *generated_key_identifier_1* into the key generate service.
- The KEY-PART AKEK key token can also be used as input to key part import service.
- To create an internal token with a specified KEY value, ICSF needs to supply a valid master key verification pattern (MKVP).

The TOKEN *key_type* changes the data encryption algorithm bits **only** on an existing DATA, IMPORTER or EXPORTER key token. Thus, if you specify TOKEN as the key_type, the only valid *rule_array* values are INTERNAL and DES, CDMF, or SYS-ENC. There is no default for the encryption algorithm selection.

Note: No pre- or post-processing or security exits are enabled for this service. No RACF checking is done, and no calls to RACF are issued when this service is used.

Use of NOCV keys is supported by ICSF. Services using NOCV keys must be processed on a CCF. This means that the key token build service will fail a request to build a key type which is not supported on the CCF if NOCV-KEK is also specified in the rule array.

The following illustrates the key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build callable services to create a control vector.

Table 22. Control Vector Generate and Key Token Build Control Vector Keyword Combinations

Key Type	**Key Usage** **Default keys are indicated in bold.** *** All keywords in the list are defaults unless one or more keywords in the list are specified.** **** The NOOFFSET keyword is only valid if NO-SPEC, IBM-PIN, GBP-PIN, or the default (NO-SPEC) is specified.**			
Notes:	Default keys are indicated in bold. * All keywords in the list are defaults unless one or more keywords in the list are specified. ** The NOOFFSET keyword is only valid if NO-SPEC, IBM-PIN, GBP-PIN, or the default (NO-SPEC) is specified.			
DATA CIPHER ENCIPHER DECIPHER MAC MACVER CVARPINE CVARENC CVARDEC CVARXCVL CVARXCVR		**SINGLE** KEYLN8 DOUBLE KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
KEYLN24	DATA			
DATAC DATAM DATAMV		**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART

Table 22. Control Vector Generate and Key Token Build Control Vector Keyword Combinations (continued)

Key Type	**Key Usage** **Default keys are indicated in bold.** __* All keywords in the list are defaults unless one or more keywords in the list are specified.__ __** The NOOFFSET keyword is only valid if NO-SPEC, IBM-PIN, GBP-PIN, or the default (NO-SPEC) is specified.__						
KEYGENKY	CLR8-ENC UKPT				**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
DKYGENKY	DDATA DMAC DMV DIMP DEXP DPVR DMKEY DMPIN DALL	DKYL0 DKYL1 DKYL2 DKYL3 DKYL4 DKYL5 DKYL6 DKYL7			**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
SECMSG	SMKEY SMPIN				**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
IKEYXLAT OKEYXLAT				**ANY** NOT-KEK DATA PIN LMTD-KEK	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
IMPORTER	OPIM* IMEX* IMIM* IMPORT*	XLATE		**ANY** NOT-KEK DATA PIN LMTD-KEK	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
EXPORTER	OPEX* IMEX* EXEX* EXPORT*	XLATE		**ANY** NOT-KEK DATA PIN LMTD-KEK	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
PINVER			__NO-SPEC**__ IBM-PIN** GBP-PIN** IBM-PINO GBP-PINO VISA-PVV INBK-PIN	NOOFFSET	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
PINGEN	CPINGEN* CPINGENA* EPINGENA* EPINGEN* EPINVER*		__NO-SPEC**__ IBM-PIN** GBP-PIN** IBM-PINO GBP-PINO VISA-PVV INBK-PIN	NOOFFSET	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART
IPINENC	CPINGENA* EPINVER* REFORMAT* TRANSLAT*				**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART

Table 22. Control Vector Generate and Key Token Build Control Vector Keyword Combinations (continued)

Key Type	**Key Usage** **Default keys are indicated in bold.** *** All keywords in the list are defaults unless one or more keywords in the list are specified.** **** The NOOFFSET keyword is only valid if NO-SPEC, IBM-PIN, GBP-PIN, or the default (NO-SPEC) is specified.**			
OPINENC	CPINENC* EPINGEN* REFORMAT* TRANSLAT*	**DOUBLE** KEYLN16 MIXED	**XPORT-OK** NO-XPORT	KEY-PART

Related Information

The ICSF key token build callable service provides a subset of the parameters and keywords available with the Transaction Security System key token build verb.

The following key types are not supported: ADATA, AMAC, CIPHERXI, CIPHERXL, CIPHERXO, UKPTBASE.

The following rule array keywords are not supported: ACTIVE, ADAPTER, CARD, CBC, CLEAR-IV, CUSP, INACTIVE, IPS, KEY-REF, MACLEN4, MACLEN6, MACLEN8, NO-IV, READER, X9.2, X9.9-1.

The *master_key_verification_number* parameter has been replaced by the *master_key_version_number* parameter. The *master_key_version_number* parameter is examined only if the KEY keyword is specified, and in this case must be zero. If KEY and INTERNAL are both specified in the rule array, the service will check for the existence of a new optional rule array keyword, MKVP. If MKVP is specified, the service will make use of the last parameter specified. Currently, this is called *masterkey_verify_parm* and is always ignored. It will now be used to contain a master key verification pattern if MKVP is specified in the *rule_array*. The service assumes the key provided by the *key_value* parameter is enciphered under the corresponding master key and will place the key into the internal token along with the verification pattern from the *masterkey_verify_parm* parameter.

The *key_register_number*, *secure_token*, and *initialization_vector* parameters are ignored.

The *pad_character* parameter must have a value of zero.

Key Translate (CSNBKTR)

The Key Translate callable service uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.

ICSF routes the Key Translate request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBKTR(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            input_key_token,
            input_KEK_key_identifier,
            output_KEK_key_identifier,
            output_key_token )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

input_key_token

Direction: Input	Type: String

A 64-byte string variable containing an external key token. The external key token contains the key to be re-enciphered (translated).

input_KEK_key_identifier

Direction: Input/Output	Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the CKDS. The internal key token contains the key-encrypting key used to decipher the key. The internal key token must contain a control vector that specifies an importer or IKEYXLAT key type. The control vector for an importer key must have the XLATE bit set to 1.

output_KEK_key_identifier

Direction: Input/Output Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the CKDS. The internal key token contains the key-encrypting key used to encipher the key. The internal key token must contain a control vector that specifies an exporter or OKEYXLAT key type. The control vector for an exporter key must have the XLATE bit set to 1.

output_key_token

Direction: Output Type: String

A 64-byte string variable containing an external key token. The external key token contains the re-enciphered key.

Restrictions

The caller must be in task mode, not in SRB mode.

Triple length DATA key tokens are not supported.

Usage Note

SAF will be invoked to check authorization to use the Key Translate service and any key labels specified as input.

Multiple Clear Key Import (CSNBCKM)

This callable service encrypts a single-, double-, or triple-length DATA key under the system master key.

Format

```
CALL CSNBCKM(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            clear_key_length,
            clear_key,
            key_identifier_length,
            key_identifier )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The *rule_array_count* parameter must be 0 or 1.

rule_array

Direction: Input	Type: String

Zero or one keyword that supplies control information to the callable service. The keyword must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks. Refer to Table 23 on page 122 for a list of keywords.

The keyword specifies the cryptographic algorithm. If no algorithm is specified, the system default algorithm is used unless a double- or triple-length DATA key is specified on a CDMF system. In this case, the resulting DATA token is marked DES.

Table 23. Keywords for Multiple Clear Key Import Rule Array Control Information

Keyword	Meaning
Algorithm (optional)	
CDMF	The output key identifier is to be a CDMF token. For a DATA key of length 16 or 24, you may not specify CDMF.
DES	The output key identifier is to be a DES token.

clear_key_length

Direction: Input Type: Integer

The *clear_key_length* specifies the length of the clear key value to import. This length must be 8, 16, or 24.

clear_key

Direction: Input Type: String

The *clear_key* specifies the clear key value to import.

key_identifier_length

Direction: Input/Output Type: Integer

The byte length of the *key_identifier* parameter. This must be at least 64 bytes.

key_identifier

Direction: Output Type: String

A 64-byte string that is to receive the internal key token. Appendix B, “Key Token Formats” on page 365 describes the key tokens.

Usage Note

This service produces an internal DATA token with a control vector which is usable on the Cryptographic Coprocessor Feature. If a valid internal token is supplied as input to the service in the *key_identifier* field, that token’s control vector will not be used in the encryption of the clear key value.

Multiple Secure Key Import (CSNBSKM)

Use this service to encipher a single-length, double-length, or triple-length key under the system master key or an importer key-encrypting key. The clear key can then be imported as any of the possible key types.

For double-length MAC keys, the importable form of the key token uses the MAC data compatibility control vector.

Only control vectors and key types supported by the Cryptographic Coprocessor Feature will be valid when importing a triple-length key.

ICSF routes the Multiple Secure Key Import request to a PCI Cryptographic Coprocessor if the control vector of a supplied internal token cannot be processed on the Cryptographic Coprocessor Feature, or if the key type is not valid for the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Format

```
CALL CSNBSKM(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            clear_key_length,
            clear_key,
            key_type,
            key_form,
            key_encrypting_key_identifier,
            imported_key_identifier_length,
            imported_key_identifier )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The *rule_array_count* parameter must be 0, 1, or 2.

rule_array

Direction: Input Type: String

Zero to two keywords that supply control information to the callable service. Each keyword must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks. The keywords are shown in Table 24.

The first keyword is the algorithm. If no algorithm is specified, the system default algorithm is used. If no algorithm is specified on a CDMF only system and either a double- or triple-length DATA key is specified, the token is marked DES. The algorithm keyword applies only when the desired output token is of key form OP and key type IMPORTER, EXPORTER, or DATA. For key form IM or any other key type, specifying DES or CDMF causes an error.

The second keyword is optional and specifies that the output key token be marked as an NOCV-KEK.

Table 24. Keywords for Multiple Secure Key Import Rule Array Control Information

Keyword	Meaning
Algorithm (optional)	
CDMF	The output key identifier is to be a CDMF token. For a DATA key of length 16 or 24, you may not specify CDMF.
DES	The output key identifier is to be a DES token.
NOCV Choice (optional)	
NOCV-KEK	The output token is to be marked as an NOCV-KEK. This keyword only applies if key form is OP and key type is IMPORTER, EXPORTER or IMP-PKA. For key form IM or any other key type, specifying NOCV-KEK causes an error.

clear_key_length

Direction: Input Type: Integer

The *clear_key_length* specifies the length of the clear key value to import. The length must be 8, 16, or 24, but cannot exceed the maximum length for the specified key type.

clear_key

Direction: Input Type: String

The *clear_key* specifies the clear key value to import.

key_type

Direction: Input Type: 8 Character String

The type of key you want to encipher under the master key or an importer key. Specify an 8-byte field that must contain a keyword from the list shown in

Table 25 or the keyword TOKEN. For types with fewer than 8 characters, the type should be padded on the right with blanks. If the key type is TOKEN, ICSF determines the key type from the control vector (CV) field in the internal key token provided in the *imported_key_identifier* parameter. If the control vector is invalid on the Cryptographic Coprocessor Feature, the multiple secure key import request will be routed to the PCI Cryptographic Coprocessor.

Table 25. Key Type Values for the Multiple Secure Key Import Callable Service

Key Type	Meaning
DATA	Data-encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAM	MAC generation key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAMV	MAC verification key. May be specified explicitly as a key type or through the TOKEN keyword.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
EXPORTER	Exporter key-encrypting key. Use this double-length key to convert any key (including a DATA key) from operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IMP-PKA	Limited authority importer key. Use this double-length key to import an encrypted RSA or DSS private key.
IPINENC	Input PIN-encrypting key. Use this double-length input key to translate PINs. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate and Encrypted PIN verify callable services.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Double-length output PIN-encrypting key. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.

key_form

Direction: Input Type: 4 Character String

The key form you want to generate. Enter a 4-byte keyword specifying whether the key should be enciphered under the master key (OP) or the importer key-encrypting key (IM). The keyword must be left-justified and padded with blanks. Valid keyword values are OP for encryption under the master key or IM for encryption under the importer key-encrypting key. If you specify IM, you must specify an importer key-encrypting key in the *key_encrypting_key_identifier* parameter. For a *key_type* of IMP-PKA, this service supports only the OP *key_form*.

key_encrypting_key_identifier

Direction: Input/Output	Type: String

A 64-byte string internal key token or key label of an importer key-encrypting key.

imported_key_identifier_length

Direction: Input/Output	Type: Integer

The byte length of the *imported_key_identifier* parameter. This must be at least 64 bytes.

imported_key_identifier

Direction: Input/Output	Type: String

A 64-byte string that is to receive the output key token. If OP is specified in the *key_form* parameter, the service returns an internal key token. If IM is specified in the *key_form* parameter, the service returns an external key token. Appendix B, “Key Token Formats” on page 365 describes the key tokens.

Note that for a DATA key of length 16 or 24, no reference will be made to the data encryption algorithm bits or to the system's default algorithm; the token will be marked DES.

Usage Notes

To generate double-length MAC and MACVER keys in the importable form, the ANSI system keys must be installed in the CKDS.

If the key to be imported is a triple-length key, only control vectors supported by the Cryptographic Coprocessor Feature are valid.

PKA Decrypt (CSNDPKD)

This callable service accepts a PKCS 1.2-formatted, wrapped key value, along with an RSA private key in either internal or external (clear) format. The external format may be a modulus-exponent form or the Chinese Remainder form of the key with clear key values. The service assumes that the sender used the corresponding RSA public key to wrap the input key value. The service unwraps the key, deformats it, and returns the deformatted value to the application in the clear. Processing for this service is routed to the PCI Cryptographic Accelerator if a PCI

Cryptographic Accelerator is online and an external clear RSA token is supplied, otherwise it is routed to the PCI Cryptographic Coprocessor or the Cryptographic Coprocessor Feature.

If an external clear key token is used, then the key values are not encrypted under any master key. PKA Decrypt with clear keys does not require any Master Keys to be installed on the Cryptographic Coprocessor Features or the PCI Cryptographic Coprocessors and PKA callable services do not have to be enabled.

The PKA decrypt callable service examines the RSA key specified in the *PKA_key_identifier* parameter to determine how to route the request. If the modulus bit length is less than 512 bits, or if the key is a X'02' form modulus-exponent private key, ICSF routes the request to the Cryptographic Coprocessor Feature. If the key is a X'08' form CRT private key or a retained private key, the service routes the request to a PCI Cryptographic Coprocessor. In the case of a retained key, the service routes the request to the specific PCI Cryptographic Coprocessor in which the key is retained. If the key is a modulus-exponent form private key with a private section ID of X'06', then the service routes the request as follows:

- Since the key must be a key-management key, if the KMMK is equal to the SMK on the Cryptographic Coprocessor Feature, the PKA decrypt service uses load balancing to route the request to either a Cryptographic Coprocessor Feature or to an available PCI Cryptographic Coprocessor.
- If the KMMK is not equal to the SMK on the Cryptographic Coprocessor Feature, the request must be processed on a PCI Cryptographic Coprocessor. If there is no PCI Cryptographic Coprocessor online, the request will fail and issue a return and reason code.

Format

```
CALL CSNDPKD(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              PKA_enciphered_keyvalue_length,
              PKA_enciphered_keyvalue,
              data_structure_length,
              data_structure,
              PKA_key_identifier_length,
              PKA_key_identifier,
              target_keyvalue_length,
              target_keyvalue)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 1.

rule_array

Direction: Input Type: String

The keyword that provides control information to the callable service. The keyword is left-justified in an 8-byte field and padded on the right with blanks.

Table 26. Keywords for PKA Decrypt

Keyword	Meaning
Recovery Method (required) specifies the method to use to recover the key value.	
PKCS-1.2	RSA DSI PKCS #1 block type 02 will be used to recover the key value.

PKA_enciphered_keyvalue_length

Direction: Input Type: integer

The length of the *PKA_enciphered_keyvalue* parameter in bytes. The maximum size that you can specify is 256 bytes.

PKA_enciphered_keyvalue

Direction: Input Type: String

This field contains the key value protected under an RSA public key. This byte-length string is left-justified within the *PKA_enciphered_keyvalue* parameter.

data_structure_length

Direction: Input	Type: Integer

The value must be 0.

data_structure

Direction: Input	Type: String

This field is currently ignored.

PKA_key_identifier_length

Direction: Input	Type: Integer

The length of the *PKA_key_identifier* parameter. When the *PKA_key_identifier* is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

PKA_key_identifier

Direction: Input	Type: String

An internal RSA private key token, the label of an internal RSA private key token, or an external RSA private key token containing a clear RSA private key in modulus-exponent or Chinese Remainder format. The corresponding public key was used to wrap the key value.

target_keyvalue_length

Direction: Input/Output	Type: Integer

The length of the *target_keyvalue* parameter. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of *target_keyvalue*.

target_keyvalue

Direction: Output	Type: String

This field will contain the decrypted, deformatted key value.

Restrictions

The exponent of the RSA public key must be odd.

Caller must be in task mode and must not be in SRB mode.

Access control checking will not be performed in the PCI Cryptographic Coprocessor when a clear external key token is supplied.

If a PKDS label name is not being supplied, then a value less than a blank (X'40') must be supplied in the first byte of the parameter or else the service fails with return code 8, reason code 11000.

Usage Notes

- The RSA private key must be enabled for key management functions.
- The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit. The service will fail with return code 12 and reason code 11020.

PKA Encrypt (CSNDPKE)

This callable service encrypts a supplied clear key value under an RSA public key. The rule array keyword specifies the format of the key prior to encryption.

This service routes requests to the Cryptographic Coprocessor Feature unless the modulus bit length of the key specified in the *PKA_key_identifier* is greater than 1024 bits.

Format

```
CALL CSNDPKE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             keyvalue_length,
             keyvalue,
             data_structure_length,
             data_structure,
             PKA_key_identifier_length,
             PKA_key_identifier,
             PKA_enciphered_keyvalue_length,
             PKA_enciphered_keyvalue)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This value must be 1.

rule_array

Direction: Input Type: String

A keyword that provides control information to the callable service. The keyword is left-justified in an 8-byte field and padded on the right with blanks.

Table 27. Keywords for PKA Encrypt

Keyword	Meaning
Formatting Method (required) specifies the method to use to format the key value prior to encryption.	
PKCS-1.2	RSA DSI PKCS #1 block type 02 format will be used to format the supplied key value.
ZERO-PAD	The key value will be padded on the left with binary zeros to the length of the PKA key modulus.

keyvalue_length

Direction: Input Type: Integer

The length of the *keyvalue* parameter. The maximum field size is 256 bytes. The actual maximum size depends on the modulus length of *PKA_key_identifier* and the formatting method you specify in the *rule_array* parameter. See Usage Notes.

keyvalue

Direction: Input Type: String

This field contains the supplied clear key value to be encrypted under the *PKA_key_identifier*.

data_structure_length

Direction: Input	Type: Integer

This value must be 0.

data_structure

Direction: Input	Type: String

This field is currently ignored.

PKA_key_identifier_length

Direction: Input	Type: Integer

The length of the *PKA_key_identifier* parameter. When the *PKA_key_identifier* is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

PKA_key_identifier

Direction: Input	Type: String

The RSA public or private key token or the label of the RSA public or private key to be used to encrypt the supplied key value.

PKA_enciphered_keyvalue_length

Direction: Input/Output	Type: integer

The length of the *PKA_enciphered_keyvalue* parameter in bytes. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of *PKA_enciphered_keyvalue*.

PKA_enciphered_keyvalue

Direction: Output	Type: String

This field contains the key value protected under an RSA public key. This byte-length string is left-justified within the *PKA_enciphered_keyvalue* parameter.

Restrictions

The exponent of the RSA public key must be odd.

The caller must be in task mode and must not be in SRB mode.

Usage Notes

- For RSA DSI PKCS #1 formatting, the key value length must be at least 11 bytes less than the modulus length of the RSA key.

- The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit. The service will fail with return code 12 and reason code 11020.
- A new *rule_array* parameter, ZERO-PAD, has been added in OS/390 R2 R10 (APAR OW48132). The key value will be padded on the left with binary zeros to the length of the PKA key modulus.

Prohibit Export (CSNBPEX)

The Prohibit Export service modifies an operational key so that it cannot be exported. The Prohibit Export service does not support NOCV key-encrypting keys, or DATA, MAC, or MACVER keys with standard control vectors (for example, control vectors supported by the Cryptographic Coprocessor Feature).

ICSF routes the Prohibit Export request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBPEX(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_identifier

Direction: Input/Output Type: String

A 64-byte string variable containing the internal key token to be modified. The returned *key_identifier* will be encrypted under the current master key.

Restriction

The caller must be in task mode, not in SRB mode.

Usage Note

SAF will be invoked to check authorization to use the Prohibit Export service.

Prohibit Export Extended (CSNBPEXX)

Use the prohibit export extended callable service to change the external token of a cryptographic key in exportable form so that it can be imported at the receiver node and is non-exportable from that node. You cannot prohibit export of DATA keys.

The inputs are an external token of the key to change in the *source_key_token* parameter and the label or internal token of the exporter key-encrypting key in the *kek_key_identifier* parameter.

CSNBPEXX is a variation of the prohibit export service CSNBPEX, which supports changing an *internal* token.

Format

```
CALL CSNBPEXX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             source_key_token,
             kek_key_identifier)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

source_key_token

Direction: Input/Output	Type: String

A 64-byte string of an external token of a key to change. It is in exportable form.

kek_key_identifier

Direction: Input/Output	Type: Integer

A 64-byte string of an internal token or label of the exporter KEK used to encrypt the key contained in the external token specified in the previous parameter.

Random Number Generate (CSNBRNG)

The callable service uses the cryptographic feature to generate a random number. The foundation for the random number generator is a time variant input with a very low probability of recycling.

Format

```
CALL CSNBRNG(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            form,
            random_number )
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

form

Direction: Input   Type: Character string

The 8-byte keyword that defines the characteristics of the random number should be left-justify and pad on the right with blanks. The keywords are listed in Table 28.

Table 28. Keywords for the Form Parameter

Keyword	Meaning
EVEN	Generate a 64-bit random number with even parity in each byte.
ODD	Generate a 64-bit random number with odd parity in each byte.
RANDOM	Generate a 64-bit random number.

Parity is calculated on the 7 high-order bits in each byte and is presented in the low-order bit in the byte.

random_number

Direction: Output   Type: String

The generated number returned by the callable service in an 8-byte variable.

Secure Key Import (CSNBSKI)

Use the secure key import callable service to encipher a single-length or double-length clear key under the system master key or under an importer key-encrypting key. The clear key can then be imported as any of the possible key types. This service does not adjust key parity.

The callable service can execute only when ICSF is in special secure mode, which is described in “Special Secure Mode” on page 9.

You can import DATA keys or KEKs whose data encryption algorithm bits indicate the encryption algorithm. DATA keys can be marked CDMF or DES. This service marks the imported DATA key token according to the system's default encryption algorithm, unless token copying overrides this. KEKs can be marked CDMF DES, or SYS-ENC. They are marked SYS-ENC unless token-copying overrides this. See the *key_identifier* parameter in this service and the key token build callable service (“Key Token Build (CSNBKTB)” on page 109) for more information.

Note: Marking of data encryption algorithm bits and token copying are performed only if the service is processed on the Cryptographic Coprocessor Feature.

To import double-length and triple-length DATA keys, or double-length MAC or MACVER keys, use the multiple secure key import (CSNBSKM) callable service. See “Multiple Secure Key Import (CSNBSKM)” on page 122.

ICSF routes the Secure Key Import request to a PCI Cryptographic Coprocessor if the control vector cannot be processed on the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Format

```
CALL CSNBSKI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            clear_key,
            key_type,
            key_form,
            importer_key_identifier,
            key_identifier )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

clear_key

Direction: Input Type: String

The clear key to be enciphered. Specify a 16-byte string (clear key value). For single-length keys, the value must be left-justified and padded with zeros. For effective single-length keys, the value of the right half must equal the value of the left half. For double-length keys, specify the left and right key values.

key_type

Direction: Input Type: Character string

The type of key you want to encipher under the master key or an importer key. Specify an 8-byte field that must contain a keyword from the list shown in Table 29 or the keyword TOKEN. If the key type is TOKEN, ICSF determines the key type from the CV in the *key_identifier* parameter. If the control vector is invalid on the Cryptographic Coprocessor Feature, the secure key import request will be routed to the PCI Cryptographic Coprocessor.

Table 29. Key Type Values for the Secure Key Import Callable Service

Key Type	Meaning
DATA	Data-encrypting key. Use this single-length, double-length, or triple-length key to encipher and decipher data.
DATAXLAT	Data translation key. Use this single-length key to reencipher text from one DATA key to another.
EXPORTER	Exporter key-encrypting key. Use this double-length key to convert any key (including a DATA key) from operational form into exportable form.
IKEYXLAT	Used to decrypt an input key in the Key Translate callable service. This is a double-length key.

Table 29. Key Type Values for the Secure Key Import Callable Service (continued)

Key Type	Meaning
IMPORTER	Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.
IMP-PKA	Limited authority importer key. Use this double-length key to import an encrypted RSA or DSS private key.
IPINENC	Input PIN-encrypting key. Use this double-length input key to translate PINs. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN translate and Encrypted PIN verify callable services.
MAC	MAC generation key. Use this single-length key to generate a message authentication code.
MACVER	MAC verification key. Use this single-length key to verify a message authentication code.
OKEYXLAT	Used to encrypt an output key in the Key Translate callable service. This is a double-length key.
OPINENC	Double-length output PIN-encrypting key. The output PIN blocks from the Encrypted PIN translate, Encrypted PIN generate, and Clear PIN generate alternate callable services are encrypted under this type of key. If an encrypted PIN block is contained in the output of the SET Block Decompose service, it may be encrypted by an OPINENC key.
PINGEN	PIN generation key. Use this double-length key to generate PINs.
PINVER	PIN verification key. Use this double-length key to verify PINs.

key_form

Direction: Input Type: Character string

The key form you want to generate. Enter a 4-byte keyword specifying whether the key should be enciphered under the master key (OP) or the importer key-encrypting key (IM). The keyword must be left-justified and padded with blanks. Valid keyword values are OP for encryption under the master key or IM for encryption under the importer key-encrypting key. If you specify IM, you must specify an importer key-encrypting key in the *importer_key_identifier* parameter. For a *key_type* of IMP-PKA, this service supports only the OP *key_form*.

importer_key_identifier

Direction: Input/Output Type: String

The importer key-encrypting key under which you want to encrypt the clear key. Specify either a 64-byte string of the internal key format or a key label. If you specify IM for the *key_form* parameter, the *importer_key_identifier* parameter is required.

key_identifier

Direction: Input/Output Type: String

The generated encrypted key. The parameter is a 64-byte string. The callable service returns either an internal key token if you encrypted the clear key under the master key (*key_form* was OP); or an external key token if you encrypted the clear key under the importer key-encrypting key (*key_form* was IM).

If the imported key_type is IMPORTER or EXPORTER and the key_form is OP, the *key_identifier* parameter changes direction to both input and output. If the application passes a valid internal key token for an IMPORTER or EXPORTER key in this parameter, the NOCV bit is propagated to the imported key token.

Note: Propagation of the NOCV bit is performed only if the service is processed on the Cryptographic Coprocessor Feature.

If the *key_type* is DATA, IMPORTER, or EXPORTER and the application passes a valid internal key token for a DATA, IMPORTER, or EXPORTER key in the *key_identifier,* this service propagates the data encryption algorithm bits to the imported key token. Propagation of token markings is only relevant when this service is processed on the Cryptographic Coprocessor Feature.

The secure key import service does not adjust key parity.

Symmetric Key Export (CSNDSYX)

Use the symmetric key export callable service to transfer an application-supplied symmetric key (a DATA key) from encryption under the DES host master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be an ICSF DES internal key token or the label of such a token in the CKDS. The symmetric key import callable service can import the PKA-encrypted form at the receiving node.

This service requires the enhanced system keys to be present in the CKDS.

ICSF routes this service to a PCI Cryptographic Coprocessor if one is available on your server. This service will not be routed to a PCI Cryptographic Coprocessor if the modulus bit length of the RSA public key is less than 512 bits.

Format

```
CALL CSNDSYX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             DATA_key_identifier_length,
             DATA_key_identifier,
             RSA_public_key_identifier_length,
             RSA_public_key_identifier,
             RSA_enciphered_key_length,
             RSA_enciphered_key)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. Value must be 1.

rule_array

Direction: Input   Type: String

Keywords that provide control information to the callable service. Table 30 lists the keywords. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage.

Table 30. Keywords for Symmetric Key Export Control Information

Keyword	Meaning
Recovery Method (required)	
PKCSOAEP	Specifies using the method found in RSA DSI PKCS #1V2 OAEP.
PKCS–1.2	Specifies using the method found in RSA DSI PKCS #1 block type 02 to recover the symmetric key.

Table 30. Keywords for Symmetric Key Export Control Information (continued)

Keyword	Meaning
ZERO-PAD	The clear key is right-justified in the field provided, and the field is padded to the left with zeroes up to the size of the RSA encryption block (which is the modulus length).

DATA_key_identifier_length

Direction: Input Type: Integer

The length of the *DATA_key_identifier* parameter. The minimum size is 64 bytes. The maximum size is 128 bytes.

DATA_key_identifier

Direction: Input/Output Type: Integer

The label or internal token of a DATA key to export for encryption under the supplied RSA public key. This service exports a DATA key of the same length as the key specified in this parameter.

RSA_public_key_identifier_length

Direction: Input Type: Integer

The length of the *RSA_public_key_identifier* parameter. The maximum size is 2500 bytes.

RSA_public_key_identifier

Direction: Input Type: String

A PKA public key token or label of the key to protect the exported symmetric key.

RSA_enciphered_key_length

Direction: Input/Output Type: Integer

The length of the *RSA_enciphered_key* parameter. This is updated with the actual length of the *RSA_enciphered_key* generated. The maximum size is 256 bytes.

RSA_enciphered_key

Direction: Output Type: String

This field contains the RSA_enciphered key, protected by the public key specified in the *RSA_public_key_identifier* field.

Restrictions

The enhanced system keys must be present in the CKDS.

Caller must be task mode and not in SRB mode.

Use of PKCSOAEP requires the PCI Cryptographic Coprocessor.

Usage Notes

This service requires that the enhanced system keys be installed in the CKDS.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit. The service will fail with return code 12 and reason code 11020.

Symmetric Key Generate (CSNDSYG)

Use the symmetric key generate callable service to generate a symmetric key (a DATA key) and return the key in two forms: DES-encrypted and encrypted under an RSA public key. (There are two types of PKA public key tokens: RSA and DSS. This callable service uses only the RSA type.) The DES encryption may be in the form of an internal token encrypted under the host DES master Key or in the external form encrypted under a key-encrypting key. You can import the PKA-encrypted form by using the symmetric key import service at the receiving node. Also use the symmetric key generate callable service to generate any importer or exporter key-encrypting key encrypted under a PKA96 RSA public key according to the PKA92 formatting structure. See “PKA92 Key Format and Encryption Process” on page 439 for more details about PKA92 formatting.

The generated internal DATA key token is marked according to the system default algorithm.

Note: Token marking is only performed if the service is processed on the Cryptographic Coprocessor Feature.

ICSF routes this service to a PCI Cryptographic Coprocessor if one is available on your server. This service will not be routed to a PCI Cryptographic Coprocessor if the modulus bit length of the RSA public key is less than 512 bits.

Format

```
CALL CSNDSYG(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_encrypting_key_identifier,
            RSA_public_key_identifier_length,
            RSA_public_key_identifier,
            DES_enciphered_key_token_length,
            DES_enciphered_key_token,
            RSA_enciphered_key_length,
            RSA_enciphered_key)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value must be 1, 2, or 3.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service. Table 31 lists the keywords. The recovery method is the method to use to recover the symmetric key. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

Table 31. Keywords for Symmetric Key Generate Control Information

Keyword	Meaning
Recovery Method (required)	
PKA92	Specifies the key-encrypting key is to be encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.

Table 31. Keywords for Symmetric Key Generate Control Information (continued)

Keyword	Meaning
PKCSOAEP	Specifies using the method found in RSA DSI PKCS #1V2 OAEP.
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02.
ZERO-PAD	The clear key is right-justified in the field provided, and the field is padded to the left with zeroes up to the size of the RSA encryption block (which is the modulus length).
Form of the DES_Enciphered_Key_Token (optional) not valid with PKA92	
EX	The DES enciphered key is enciphered by an EXPORTER key that is provided through the *key_encrypting_key_identifier* parameter.
IM	The DES enciphered key is enciphered by an IMPORTER key that is provided through the *key_encrypting_key_identifier* parameter. This form requires the enabling of Special Secure Mode. Special Secure Mode is not required if a PCI Cryptographic Coprocessor is available and the modulus bit length is greater than or equal to 512 bits.
OP	The DES enciphered key is enciphered by the master key. The *key_encrypting_key_identifier* parameter is ignored. This is the default.
DES Key Length (optional)	
DOUBLE	Generates a double-length DES key.
KEYLN8	Generates a single-length DES key. This is the default.
KEYLN16	Generates a double-length DES DATA key.
KEYLN24	Generates a triple-length DES DATA key.
SINGLE	Generates a single-length DES key.
SINGLE-R	Generates a key-encrypting key that has equal left and right halves allowing it to perform as a single-length key. Valid only for the recovery method of PKA92.

key_encrypting_key_identifier

Direction: Input/Output　　Type: String

The label or internal token of a key-encrypting key. If the *rule_array* specifies IM, this DES key must be an IMPORTER. If the *rule_array* specifies EX, this DES key must be an EXPORTER.

RSA_public_key_identifier_length

Direction: Input　　Type: Integer

The length of the *RSA_public_key_identifier* parameter. If the *RSA_public_key_identifier* parameter is a label, this parameter specifies the length of the label. The maximum size is 2500 bytes.

RSA_public_key_identifier

Direction: Input　　Type: String

The token, or label, of the RSA public key to be used for protecting the generated symmetric key.

DES_enciphered_key_token_length

Direction: Input/Output	Type: Integer

The length of the *DES_enciphered_key_token*. This field is updated with the actual length of the *DES_enciphered_key_token* that is generated. The minimum size is 64 bytes. The maximum size is 128 bytes.

DES_enciphered_key_token

Direction: Input/Output	Type: String

This parameter contains the generated DES-enciphered DATA key in the form of an internal or external token, depending on *rule_array* specification.

RSA_enciphered_key_length

Direction: Input/Output	Type: Integer

The length of the *RSA_enciphered_key* parameter. This service updates this with the actual length of the *RSA_enciphered_key* it generates. The maximum size is 256 bytes.

RSA_enciphered_key

Direction: Input/Output	Type: String

This field contains the RSA enciphered key, which the public key specified in the *RSA_public_key_identifier* field protects. If you specify PKA92, on input specify an internal (operational) DES key token.

Restrictions

If you specify IM in the *rule_array*, you must enable Special Secure Mode.

Special Secure Mode is not required if a PCI Cryptographic Coprocessor is available and the modulus bit length of the RSA public key is greater than or equal to 512 bits.

The caller must be in task mode and not in SRB mode.

Use of PKA92 requires the PCI Cryptographic Coprocessor.

| Use of PKCSOAEP requires the PCI Cryptographic Coprocessor.

Usage Notes

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit. The service will fail with return code 12 and reason code 11020.

Specification of PKA92 with an input NOCV key-encrypting key token is not supported.

Use the PKA92 key-formatting method to generate a key-encrypting key. The service enciphers one key copy using the key encipherment technique employed in the IBM Transaction Security System (TSS) 4753, 4755, and AS/400 cryptographic product PKA92 implementations (see “PKA92 Key Format and Encryption Process” on page 439). The control vector for the RSA-enciphered copy of the key is taken from an internal (operational) DES key token that must be present on input in the *RSA_enciphered_key* variable. Only key-encrypting keys that conform to the rules for an OPEX case under the key generate service are permitted. The control vector for the local key is taken from a DES key token that must be present on input in the *DES_enciphered_key_token* variable. The control vector for one key copy must be from the EXPORTER class while the control vector for the other key copy must be from the IMPORTER class.

Symmetric Key Import (CSNDSYI)

Use the symmetric key import callable service to import a symmetric (DES) DATA key enciphered under an RSA public key. (There are two types of PKA private key tokens: RSA and DSS. This callable service uses only the RSA type.) It returns the key in operational form, enciphered under the master key. It marks the key according to the system's default encryption algorithm, unless token copying overrides this. Marking of data encryption algorithm bits and token copying are only performed if the service is executed on the Cryptographic Coprocessor Feature. The symmetric key import service also supports import of a PKA92-formatted DES key-encrypting key under a PKA96 RSA public key.

The symmetric key import callable service examines the RSA key specified in the *RSA_private_key_identifier* parameter to determine how to route the request. If the key is a modulus-exponent form private key with a private section ID of X'02', ICSF routes the request to the Cryptographic Coprocessor Feature. If the token modulus bit length is less than 512, the request will also be routed to the Cryptographic Coprocessor Feature. In either of these cases, if the PKA92 recovery method is specified, the request will fail. If the key is a modulus-exponent form private key with a private section ID of X'06', a CRT form private key with a section ID of X'08', or a retained private key, ICSF routes the request to a PCI Cryptographic Coprocessor. In the case of a retained key, the service routes the request to the specific PCI Cryptographic Coprocessor in which the key is retained. If there is no PCI Cryptographic Coprocessor online:

- and the *RSA_private_key_identifier* is a retained private key or a CRT form private key, the request will fail and a return and reason code will be issued
- and the KMMK is equal to the SMK on the Cryptographic Coprocessor Feature, ICSF routes the request to a Cryptographic Coprocessor Feature
- and the KMMK is not equal to the SMK on the Cryptographic Coprocessor Feature, the request must be processed on a PCI Cryptographic Coprocessor. Since there is no PCI Cryptographic Coprocessor online, the request will fail and a return code and reason code will be issued.
- and the PKA92 recovery method is specified, the request will fail.

Format

```
CALL CSNDSYI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            RSA_enciphered_key_length,
            RSA_enciphered_key,
            RSA_private_key_identifier_length,
            RSA_private_key_identifier,
            target_key_identifier_length,
            target_key_identifier)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input Type: String

The keyword that provides control information to the callable service. Table 32 provides a list. The recovery method is the method to use to recover the symmetric key. The keyword is left-justified in an 8-byte field and padded on the right with blanks.

Table 32. Keywords for Symmetric Key Import Control Information

Keyword	Meaning
Recovery Method (required)	
PKA92	Specifies the key-encrypting key is encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.
PKCSOAEP	Specifies using the method found in RSA DSI PKCS #1V2 OAEP.
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02.
ZERO-PAD	The clear key is right-justified in the field provided, and the field is padded to the left with zeroes up to the size of the RSA encryption block (which is the modulus length).

RSA_enciphered_key_length

Direction: Input Type: integer

The length of the *RSA_enciphered_key* parameter. The maximum size is 256 bytes.

RSA_enciphered_key

Direction: Input Type: String

The key to import, protected under an RSA public key. The encrypted key is in the low-order bits (right-justified) of a string whose length is the minimum number of bytes that can contain the encrypted key. This string is left-justified within the *RSA_enciphered_key* parameter.

RSA_private_key_identifier_length

Direction: Input Type: Integer

The length of the *RSA_private_key_identifier* parameter. When the *RSA_private_key_identifier* parameter is a key label, this field specifies the length of the label. The maximum size is 2500 bytes.

RSA_private_key_identifier

Direction: Input Type: String

An internal RSA private key token or label whose corresponding public key protects the symmetric key.

target_key_identifier_length

Direction: Input/Output Type: Integer

The length of the *target_key_identifier* parameter. This field is updated with the actual length of the *target_key_identifier* that is generated. The size must be 64 bytes.

target_key_identifier

Direction: Input/Output Type: String

This field contains the internal token of the imported symmetric key. It is marked according to the system default algorithm, CDMF or DES. If you supply as input a valid internal token of an 8-byte DATA key, this service propagates the algorithm markings from the supplied token to the imported DATA key. If the key length is 16 or 24, however, the key token is marked as DES. Propagation of token markings is only relevant when this service is processed on the Cryptographic Coprocessor Feature.

Except for PKA92 processing, this service produces a DATA key token with a key of the same length as that contained in the imported token.

Restrictions

The exponent of the RSA public key must be odd.

The caller must be in task mode and not in SRB mode.

Use of PKA92 requires the optional PCI Cryptographic Coprocessor.

| Use of PKCSOAEP requires the PCI Cryptographic Coprocessor.

Usage Notes

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit. The service will fail with return code 12 and reason code 11020.

Specification of PKA92 with an input NOCV key-encrypting key token is not supported.

During initialization of a PCI Cryptographic Coprocessor, an Environment Identification, or EID, of zero will be set in the coprocessor. This will be interpreted by the PKA Symmetric Key Import service to mean that environment identification checking is to be bypassed. Thus it is possible on a OS/390 system for a key-encrypting key RSA-enciphered at a node (EID) to be imported at the same node.

Transform CDMF Key (CSNBTCK)

Use the transform CDMF key callable service to change a CDMF DATA key in an internal or external token to a transformed shortened DES key. You can also use the key label of a CKDS record as input. This callable service is implemented on S/390 Enterprise Servers and S/390 Multiprise. The Cryptographic Coprocessor Feature on S/390 Enterprise Servers and S/390 Multiprise is configured as either CDMF or DES-CDMF. This callable service ignores the input internal DATA token markings, and it marks the output internal token for use in the DES.

If the input DATA key is in an external token, the operational KEK must be marked as DES or SYS-ENC. The service fails for an external DATA key encrypted under a KEK that is marked as CDMF.

Format

```
CALL CSNBTCK(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            source_key_identifier,
            kek_key_identifier,
            target_key_identifier )
```

Parameters

return_code

Direction: Output

Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output

Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output

Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output

Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. This number must be 0.

rule_array

Direction: Input	Type: String

Currently no *rule_array* keywords are defined for this service, but you still must specify this parameter.

source_key_identifier

Direction: Input/Output	Type: String

A 64-byte string of the internal token, external token or key label that contains the DATA key to transform. Token markings on this key token are ignored.

kek_key_identifier

Direction: Input/Output	Type: String

A 64-byte string of the internal token or a key label of a key encrypting key under which the *source_key_identifier* is encrypted.

Note: If you supply a label for this parameter, the label must be unique in the CKDS.

target_key_identifier

Direction: Output	Type: String

A 64-byte string where the internal token or external token of the transformed shortened DES key is returned. The internal token is marked as DES.

Restrictions

This service is available on S/390 Enterprise Servers and S/390 Multiprise with Cryptographic Coprocessor Features. These systems may be configured as either CDMF or DES-CDMF.

Usage Notes

This service transforms a CDMF DATA key to a transformed shortened DES DATA key to allow interoperability to a DES-only capable system. The algorithm is described in Transform CDMF Key Algorithm.

User Derived Key (CSFUDK)

Use the user derived key callable service to generate a single-length or double-length MAC key or to update an existing user derived key. A single-length MAC key can be used to compute a MAC following the ANSI X9.9, ANSI X9.19, or the Europay, MasterCard and VISA (EMV) Specification MAC processing rules. A double-length MAC key can be used to compute a MAC following either the ANSI X9.19 optional double MAC processing rule or the EMV Specification MAC processing rule.

This service updates an existing user derived key by XORing it with data you supply in the *data_array* parameter. This is called SESSION MAC key generation by VISA.

This service adjusts the user derived key or SESSION MAC key to odd parity. The parity of the supplied derivation key is not tested.

Format

```
CALL CSFUDK(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_type,
            rule_array_count,
            rule_array,
            derivation_key_identifier,
            source_key_identifier,
            data_array,
            generated_key_identifier)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_type

Direction: Input Type: String

The 8-byte keyword of ’MAC ’ or ’MACD ’ that specifies the key type to be generated. The keyword must be left-justified and padded on the right with blanks. MAC specifies an 8-byte, single-length MAC key which is used in the ANSI X9.9-1 or the ANSI X9.19 basic MAC processing rules. MACD specifies a 16-byte, double-length internal MAC key that uses the single-length control vector for both the left and right half of the key (MAC ∥ MAC). The double-length MAC key is used in the ANSI X9.19 optional double-key MAC processing rules. The keyword ’TOKEN ’ is also accepted. If you specify TOKEN with a *rule_array* of VISA or NOFORMAT, the key type is determined by the valid internal token of the single-length or double-length MAC key in the *generated_key_identifier* parameter. If you specify TOKEN with a *rule_array* of SESS-MAC, the key type is determined by the valid internal token of the single-length or double-length MAC key in the *source_key_identifier.*

rule_array_count

Direction: Input Type: Integer

The number of keywords specified in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input Type: Character string

The process rule for the user derived key in an 8-byte field. The keywords must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks. For example,

```
'VISA    '
```

The keywords are shown in Table 33.

Table 33. Keywords for User Derived Key Control Information

Keyword	Meaning
User Derived Key Process Rules (required)	
NOFORMAT	For generating a user derived key with no formatting done on the array before encryption under the *derivation_key_identifier.*

Table 33. Keywords for User Derived Key Control Information (continued)

Keyword	Meaning
SESS-MAC	To update an existing user derived key supplied in the *source_key_ identifier* parameter with data provided in the *data_array* parameter.
VISA	For generating a user derived key using the VISA algorithm to format the data array input before encryption under the *derivation_key_identifier*. For guidance information refer to the VISA Integrated Circuit Card Specification, V1.3 Aug 31, 1996.

derivation_key_identifier

Direction: Input/Output Type: String

For a *rule_array* value of VISA or NOFORMAT, this is a 64-byte key label or internal key token of the derivation key used to generate the user derived key. The key must be an EXPORTER key type. For any other keyword, this field must be a null token.

source_key_identifier

Direction: Input/Output Type: String

For a *rule_array* value of SESS-MAC, this is a 64-byte internal token of a single-length or double-length MAC key. For any other keyword, this field must be a null token.

data_array

Direction: Input Type: String

Two 16-byte data elements required by the corresponding *rule_array* and *key_type* parameters. The data array consists of two 16-byte hexadecimal character fields whose specification depends on the process rule and key type. VISA requires only one 16-byte hexadecimal character input. Both NOFORMAT and SESS-MAC require one 16-byte input for a key type of MAC and two 16-byte inputs for a key type of MACD. If only one 16-byte field is required, then the rest of the data array is ignored by the callable service.

generated_key_identifier

Direction: Input/Output Type: String

The 64-byte internal token of the generated single-length or double-length MAC key. This is an input field only if TOKEN is specified for *key_type*.

Usage Note

This service requires that the ANSI system keys be installed in the CKDS.

Chapter 5. Protecting Data

Use ICSF to protect sensitive data stored on your system, sent between systems, or stored off your system on magnetic tape. To protect data, encipher it under a key. When you want to read the data, decipher it from ciphertext to plaintext form.

ICSF provides *encipher* and *decipher callable services* to perform these functions. If you use a key to encipher data, you must use the same key to decipher the data. To use clear keys directly, ICSF provides *encode* and *decode callable services*. These services encipher and decipher with clear keys. You can use clear keys indirectly by first using the clear key import callable service, and then using the encipher and decipher callable services.

This chapter describes the following services:

- "Ciphertext Translate (CSNBCTT and CSNBCTT1)" on page 159
- "Decipher (CSNBDEC and CSNBDEC1)" on page 162
- "Decode (CSNBDCO)" on page 168
- "Encipher (CSNBENC and CSNBENC1)" on page 170
- "Encode (CSNBECO)" on page 177
- "Symmetric Key Decipher (CSNBSYD)" on page 178
- "Symmetric Key Encipher (CSNBSYE)" on page 183

Modes of Operation

To encipher or decipher data or keys, ICSF uses either the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES) algorithm or the Commercial Data Masking Facility (CDMF). The DES algorithm is documented in *Federal Information Processing Standard #46*. CDMF provides DES cryptography using an effectively shortened DATA key. See "System Encryption Algorithm" on page 26 for more information.

To encipher or decipher data, ICSF also uses the U.S. National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) algorithm. The AES algorithm is documented in a draft Federal Information Processing Standard.

ICSF enciphers and deciphers using the following modes of operation:

- *Cipher block chaining (CBC)*
- *Electronic code book (ECB)*

Cipher Block Chaining (CBC) Mode

The CBC mode uses an initial chaining vector (ICV) in its processing. The CBC mode only processes blocks of data in exact multiples of eight. The ICV is exclusive ORed with the first 8 bytes of plaintext before the encryption step; the 8-byte block of ciphertext just produced is exclusive ORed with the next 8-byte block of plaintext, and so on. You must use the same ICV to decipher the data. This disguises any pattern that may exist in the plaintext. ICSF uses the CBC encipherment mode for encrypting and decrypting data using the encipher and decipher callable services.

Electronic Code Book (ECB) Mode

In the ECB mode, each 64-bit block of plaintext is separately enciphered and each block of the ciphertext is separately deciphered. In other words, the encipherment or decipherment of a block is totally independent of other blocks. ICSF uses the ECB encipherment mode for enciphering and deciphering data with clear keys using the encode and decode callable services.

ICSF does not support ECB encipherment mode on CDMF-only systems.

Triple DES Encryption

Triple-DES encryption uses a triple-length DATA key comprised of three 8-byte DES keys to encipher 8 bytes of data using the following method:

- Encipher the data using the first key
- Decipher the result using the second key
- Encipher the second result using the third key

The procedure is reversed to decipher data that has been triple-DES enciphered:

- Decipher the data using the third key
- Encipher the result using the second key
- Decipher the second result using the first key

ICSF uses the triple-DES encryption in the CBC encipherment mode.

A variation of the triple DES algorithm supports the use of a double-length DATA key comprised of two 8-byte DATA keys. In this method, the first 8-byte key is reused in the last encipherment step.

Triple-DES encryption is available only on the S/390 G4 Enterprise Server (with LIC driver 98), or above. Due to export regulations, triple-DES encryption may not be available on your processor.

Processing Rules

ICSF handles this chaining for each 8-byte block of data, from the first block until the last complete 8-byte block of data in each encipher call. There are different types of *processing rules* you can choose for cipher block chaining. You choose the type of processing rule that the callable service should use for CBC mode:

- **Cipher block chaining (CBC).** In exact multiples of 8 bytes.
- **Cryptographic Unit Support Program (CUSP).** Not necessarily in exact multiples of 8 bytes. The ciphertext is the same length of the plaintext.
- **Information Protection System (IPS).** Not necessarily in exact multiples of 8 bytes. The ciphertext is the same length of the plaintext.
- **ANSI X9.23.** Not necessarily in exact multiples of 8 bytes. This processing rule pads the plaintext so that the ciphertext produced is in exact multiples of 8 bytes.
- **IBM 4700.** Not necessarily in exact multiples of 8 bytes. This processing rule pads the plaintext so that the ciphertext produced is in exact multiples of 8 bytes.

Cipher Processing Rules describes the cipher processing rules in detail.

The resulting chaining value, after an encipher call, is known as an *output chaining vector (OCV)*. When there are multiple cipher requests, the application can pass the output chaining vector from the previous encipher call as the ICV in the next encipher call. This produces chaining between successive calls, which is known as *record chaining*. ICSF provides the ICV selection keyword CONTINUE in the *rule_array* parameter that an application can use to select record chaining with the CBC, IPS, and CUSP processing rules.

A chaining vector allows you to simulate CUSP or IPS record chaining by calculating the correct OCV. To do either the CUSP or IPS method of record chaining in the encipher and decipher callable services, the OCV from one service invocation is passed as the initialization vector to the next invocation. An OCV is produced for all processing rules. The OCV is the leftmost 8 bytes of the *chaining_vector* parameter.

Ciphertext Translate (CSNBCTT and CSNBCTT1)

ICSF provides a ciphertext translate callable service on DES-capable systems. The callable service deciphers encrypted data (ciphertext) under one data translation key and reenciphers it under another data translation key without having the data appear in the clear outside the Integrated Cryptographic Feature. ICSF uses the data translation key as either the input or the output data transport key. Such a function is useful in a multiple node network, where sensitive data is passed through multiple nodes before it reaches its final destination.

“Using the Ciphertext Translate Callable Service” on page 39 provides some tips on using the callable service.

Use the ciphertext translate callable service to decipher text under an “input” key and then to encipher the text under an “output” key. The callable service uses the cipher block chaining (CBC) mode of the DES. This service is available only on a DES-capable system.

Choosing Between CSNBCTT and CSNBCTT1

CSNBCTT and CSNBCTT1 provide identical functions. When choosing the service to use, consider the following:

- **CSNBCTT** requires the input text and output text to reside in the caller’s primary address space. Also, a program using CSNBCTT adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBCTT1** allows the input text and output text to reside either in the caller’s primary address space or in a data space. This allows you to translate more data with one call. However, a program using CSNBCTT1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBCTT1, *text_id_in* and *text_id_out* are access list entry token (ALET) parameters of the data spaces containing the input text and output text.

Format

```
CALL CSNBCTT(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_identifier_in,
            key_identifier_out,
            text_length,
            text_in,
            initialization_vector_in,
            initialization_vector_out,
            text_out )
```

Ciphertext Translate (CSNBCTT and CSNBCTT1)

```
CALL CSNBCTT1(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier_in,
              key_identifier_out,
              text_length,
              text_in,
              initialization_vector_in,
              initialization_vector_out,
              text_out,
              text_id_in,
              text_id_out )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

key_identifier_in

Direction: Input/Output	Type: String

The 64-byte string of the internal key token containing the data translation (DATAXLAT) key, or the label of the CKDS record containing the DATAXLAT key used to encipher the input string.

key_identifier_out

Direction: Input/Output Type: String

The 64-byte string of an internal key token containing the DATAXLAT key, or the label of the CKDS record containing the DATAXLAT key, used to reencipher the encrypted text.

text_length

Direction: Input Type: Integer

The length of the ciphertext that is to be processed. The text length must be a multiple of 8 bytes. The MAXLEN keyword in the options file determines the upper limit of the length of the text.

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

text_in

Direction: Input Type: String

The text that is to be translated. The text is enciphered under the data translation key specified in the *key_identifier_in* parameter.

initialization_vector_in

Direction: Input Type: String

The 8-byte initialization vector that is used to decipher the input data. This parameter is the initialization vector used at the previous cryptographic node.

initialization_vector_out

Direction: Input Type: String

The 8-byte initialization vector that is used to encipher the input data. This is the new initialization vector used when the callable service enciphers the plaintext.

text_out

Direction: Output Type: String

The field where the callable service returns the translated text.

text_id_in

Direction: Input Type: Integer

For CSNBCTT1 only, the ALET of the text to be translated.

text_id_out

Direction: Input	Type: Integer

For CSNBCTT1 only, the ALET of the *text_out* field that the application supplies.

Restrictions

The input ciphertext length must be an exact multiple of 8 bytes. The minimum length of the ciphertext that can be translated is 8 bytes.

You cannot use this service on a CDMF-only system.

Usage Note

The initialization vectors must have already been established between the communicating applications or must be passed with the data.

Decipher (CSNBDEC and CSNBDEC1)

Use the decipher callable service to decipher data in an address space or a data space using the cipher block chaining mode. ICSF supports the following processing rules to decipher data. You choose the type of processing rule that the decipher callable service should use for block chaining.

Processing Rule	Purpose
ANSI X9.23	For cipher block chaining. The ciphertext must be an exact multiple of 8 bytes, but the plaintext will be 1 to 8 bytes shorter than the ciphertext. The *text_length* will also be reduced to show the original length of the plaintext.
CBC	For cipher block chaining. The ciphertext must be an exact multiple of 8 bytes, and the plaintext will have the same length.
CUSP	For cipher block chaining, but the ciphertext can be of any length. The plaintext will be the same length as the ciphertext.
IBM 4700	For cipher block chaining. The ciphertext must be an exact multiple of 8 bytes, but the plaintext will be 1 to 8 bytes shorter than the ciphertext. The *text_length* will also be reduced to show the original length of the plaintext.
IPS	For cipher block chaining, but the ciphertext can be of any length. The plaintext will be the same length as the ciphertext.

The cipher block chaining (CBC) mode uses an initial chaining value (ICV) in its processing. The first 8 bytes of ciphertext is deciphered and then the ICV is exclusive ORed with the resulting 8 bytes of data to form the first 8-byte block of plaintext. Thereafter, the 8-byte block of ciphertext is deciphered and exclusive ORed with the previous 8-byte block of ciphertext until all the ciphertext is deciphered.

The selection between single-DES decryption mode and triple-DES decryption mode is controlled by the length of the key supplied in the *key_identifier* parameter. If a single-length key is supplied, single-DES decryption is performed. If a double-length or triple-length key is supplied, triple-DES decryption is performed.

A different ICV may be passed on each call to the decipher callable service. However, the same ICV that was used in the corresponding encipher callable service must be passed.

Short blocks are text lengths of 1 to 7 bytes. A short block can be the only block. Trailing short blocks are blocks of 1 to 7 bytes that follow an exact multiple of 8 bytes. For example, if the text length is 21, there are two 8-byte blocks and a trailing short block of 5 bytes. Because the DES and CDMF process only text in exact multiples of 8 bytes, some special processing is required to decipher such short blocks. Short blocks and trailing short blocks of 1 to 7 bytes of data are processed according to the Cryptographic Unit Support Program (CUSP) rules, or by the record chaining scheme devised by and used in the Information Protection System (IPS) in the IPS/CMS product.

These methods of treating short blocks and trailing short blocks do not increase the length of the ciphertext over the plaintext. If the plaintext was *padded* during encipherment, the length of the ciphertext will always be an exact multiple of 8 bytes.

ICSF supports the following padding schemes:

- ANSI X9.23
- 4700-PAD

Choosing Between CSNBDEC and CSNBDEC1

CSNBDEC and CSNBDEC1 provide identical functions. When choosing which service to use, consider the following:

- **CSNBDEC** requires the ciphertext and plaintext to reside in the caller's primary address space. Also, a program using CSNBDEC adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBDEC1** allows the ciphertext and plaintext to reside either in the caller's primary address space or in a data space. This can allow you to decipher more data with one call. However, a program using CSNBDEC1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBDEC1, *cipher_text_id* and *clear_text_id* are access list entry token (ALET) parameters of the data spaces containing the ciphertext and plaintext.

Format

```
CALL CSNBDEC(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_identifier,
            text_length,
            cipher_text,
            initialization_vector,
            rule_array_count,
            rule_array,
            chaining_vector,
            clear_text )
```

```
CALL CSNBDEC1(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_identifier,
            text_length,
            cipher_text,
            initialization_vector,
            rule_array_count,
            rule_array,
            chaining_vector,
            clear_text,
            cipher_text_id,
            clear_text_id )
```

Parameters

return_code

Direction: Output

Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output

Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output

Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

key_identifier

Direction: Input/Output	Type: String

A 64-byte string that is the internal key token containing the data-encrypting key, or the label of a CKDS record containing a data-encrypting key, to be used for deciphering the data. If the key token or key label contains a single-length key, single-DES decryption is performed. If the key token or key label contains a double-length or triple-length key, triple-DES decryption is performed.

text_length

Direction: Input/Output	Type: Integer

On entry, you supply the length of the ciphertext. The MAXLEN keyword in the options file, as provided by the installation, determines the upper limit of the length of the text. A zero value for the *text_length* parameter is not valid. If the returned deciphered text (*clear_text* parameter) is a different length because of the removal of padding bytes, the value is updated to the length of the plaintext.

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

The application program passes the length of the ciphertext to the callable service. The callable service returns the length of the plaintext to your application program.

cipher_text

Direction: Input	Type: String

The text to be deciphered.

initialization_vector

Direction: Input	Type: String

The 8-byte supplied string for the cipher block chaining. The first block of the ciphertext is deciphered and exclusive ORed with the initial chaining vector (ICV) to get the first block of cleartext. The input block is the next ICV. To decipher the data, you must use the same ICV used when you enciphered the data.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supply in the *rule_array* parameter. The value must be 1, 2, or 3.

rule_array

Direction: Input Type: Character string

An array of 8-byte keywords providing the processing control information. The array is positional. See the keywords in Table 34. The first keyword in the array is the processing rule. You choose the processing rule you want the callable service to use for deciphering the data. The second keyword is the ICV selection keyword. The third keyword (or the second if the ICV selection keyword is allowed to default) is the encryption algorithm to use.

The service will fail if keyword DES is specified in the *rule_array* in a CDMF-only system. The service will likewise fail if keyword CDMF is specified in the *rule_array* in a DES-only system.

Table 34. Keywords for the Decipher Rule Array Control Information

Keyword	Meaning
Processing Rule (required)	
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of 8 bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.
CUSP	Performs deciphering that is compatible with IBM's CUSP and PCF products. The data can be of any length and does not need to be in multiples of 8 bytes. The ciphertext will be the same length as the plaintext. The CUSP/PCF OCV is placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the CUSP/PCF OCV from the previous call is used as the ICV for this call.
IPS	Performs deciphering that is compatible with IBM's IPS product. The data can be of any length and does not need to be in multiples of 8 bytes. The ciphertext will be the same length as the plaintext. The IPS OCV is placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the IPS OCV from the previous call is used as the ICV for this call.
X9.23	Deciphers with cipher block chaining and text length reduced to the original value. This is compatible with the requirements in ANSI standard X9.23. The ciphertext length must be an exact multiple of 8 bytes. Padding is removed from the plaintext.
4700-PAD	Deciphers with cipher block chaining and text length reduced to the original value. The ciphertext length must be an exact multiple of 8 bytes. Padding is removed from the plaintext.
ICV Selection (optional)	

Table 34. Keywords for the Decipher Rule Array Control Information (continued)

Keyword	Meaning
CONTINUE	This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. CONTINUE is valid only for processing rules CBC, IPS, and CUSP.
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. INITIAL is the default value.
Encryption Algorithm (optional)	
CDMF	This specifies using the Commercial Data Masking Facility and ignoring the token marking. You cannot use double- or triple-length keys with CDMF.
DES	This specifies using the data encryption standard and ignoring the token marking.
TOKEN	This specifies using the data encryption algorithm in the DATA key token. This is the default.

Cipher Processing Rules describes the cipher processing rules in detail.

chaining_vector

Direction: Input/Output Type: String

An 18-byte field that ICSF uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first 8 bytes in the 18-byte string.

The direction is output if the ICV selection keyword of the *rule_array* parameter is INITIAL. The direction is input/output if the ICV selection keyword of the *rule_array* parameter is CONTINUE.

clear_text

Direction: Output Type: String

The field where the callable service returns the deciphered text.

cipher_text_id

Direction: Input Type: Integer

For CSNBDEC1 only, the ALET of the ciphertext to be deciphered.

clear_text_id

Direction: Input Type: Integer

For CSNBDEC1 only, the ALET of the clear text supplied by the application.

Restrictions

The service will fail under the following conditions:

- If the keyword DES is specified in the *rule_array* parameter in a CDMF-only system
- If the keyword CDMF is specified in the *rule_array* parameter in a DES-only system
- If the key token contains double or triple-length keys and triple-DES is not enabled.

Usage Note

Only a DATA key token or DATA key label can be used in this service.

Related Information

You **cannot** overlap the plaintext and ciphertext fields. For example:

```
pppppp
     cccccc  is incorrect.

cccccc
     pppppp  is incorrect.

ppppppcccccc is correct.

P represents the plaintext and c represents the ciphertext.
```

Cipher Processing Rules discusses the cipher processing rules.

The encipher callable services (CSNBENC and CSNBENC1) are described under “Encipher (CSNBENC and CSNBENC1)” on page 170.

Decode (CSNBDCO)

Use the decode callable service (CSNBDCO) to decipher an 8-byte string using a clear key. The callable service uses the electronic code book (ECB) mode of the DES. (This service is available only on a DES-capable system.)

Considerations

If you have only a clear key, you are *not* limited to using only the encode and decode callable services. You can pass your clear key to the clear key import service, and get back a token that will allow you to use the encipher and decipher callable services.

Format

```
CALL CSNBDCO(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             clear_key,
             cipher_text,
             clear_text)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

clear_key

Direction: Input	Type: String

The 8-byte clear key value that is used to decode the data.

cipher_text

Direction: Input	Type: String

The ciphertext that is to be decoded. Specify 8 bytes of text.

clear_text

Direction: Output	Type: String

The 8-byte field where the plaintext is returned by the callable service.

Restriction

You cannot use this service on a CDMF-only system.

Encipher (CSNBENC and CSNBENC1)

Use the encipher callable service to encipher data in an address space or a data space using the cipher block chaining mode. ICSF supports the following processing rules to encipher data. You choose the type of processing rule that the encipher callable service should use for the block chaining.

Processing Rule	Purpose
ANSI X9.23	For block chaining not necessarily in exact multiples of 8 bytes. This process rule pads the plaintext so that ciphertext produced is an exact multiple of 8 bytes.
CBC	For block chaining in exact multiples of 8 bytes.
CUSP	For block chaining not necessarily in exact multiples of 8 bytes. The ciphertext will be the same length as the plaintext.
IBM 4700	For block chaining not necessarily in exact multiples of 8 bytes. This process rule pads the plaintext so that the ciphertext produced is an exact multiple of 8 bytes.
IPS	For block chaining not necessarily in exact multiples of 8 bytes. The ciphertext will be the same length as the plaintext.

For more information about the processing rules, see Table 35 on page 174 and Cipher Processing Rules.

The cipher block chaining (CBC) mode of operation uses an initial chaining vector (ICV) in its processing. The ICV is exclusive ORed with the first 8 bytes of plaintext before the encryption step, and thereafter, the 8-byte block of ciphertext just produced is exclusive ORed with the next 8-byte block of plaintext, and so on. This disguises any pattern that may exist in the plaintext.

The selection between single-DES encryption mode and triple-DES encryption mode is controlled by the length of the key supplied in the *key_identifier* parameter. If a single-length key is supplied, single-DES encryption is performed. If a double-length or triple-length key is supplied, triple-DES encryption is performed.

To nullify the CBC effect on the first 8-byte block, supply 8 bytes of zero. However, the ICV may require zeros.

Cipher block chaining also produces a resulting chaining value called the output chaining vector (OCV). The application can pass the OCV as the ICV in the next encipher call. This results in *record chaining*.

Note that the OCV that results is the same, whether an encipher or a decipher callable service was invoked, assuming the same text, ICV, and key were used.

Short blocks are text lengths of 1 to 7 bytes. A short block can be the only block. Trailing short blocks are blocks of 1 to 7 bytes that follow an exact multiple of 8 bytes. For example, if the text length is 21, there are two 8-byte blocks, and a trailing short block of 5 bytes. Short blocks and trailing short blocks of 1 to 7 bytes of data are processed according to the Cryptographic Unit Support Program (CUSP) rules, or by the record chaining scheme devised by and used by the

Information Protection System (IPS) in the IPS/CMS program product. These methods of treating short blocks and trailing short blocks do not increase the length of the ciphertext over the plaintext.

An alternative method is to pad the plaintext and produce a ciphertext that is longer than the plaintext. The plaintext can be padded with up to 8 bytes using one of several padding schemes. This padding produces a ciphertext that is an exact multiple of 8 bytes long.

If the ciphertext is to be transmitted over a network, where one or more intermediate nodes will use the ciphertext translate callable service, the ciphertext *must* be produced using one of the following methods of padding:

- ANSI X9.23
- 4700

If the cleartext is already a multiple of 8, the ciphertext can be created using any processing rule.

Because of padding, the returned ciphertext length is longer than the provided plaintext; the *text_length* parameter *will have been modified*. The returned ciphertext field should be 8 bytes longer than the length of the plaintext to accommodate the maximum amount of padding. You should provide this extension in your installation's storage because ICSF cannot detect whether the extension was done.

The minimum length of data that can be enciphered is one byte. Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

Attention: If you lose the data-encrypting key under which the data (plaintext) is enciphered, the data enciphered under that key (ciphertext) **cannot** be recovered.

Choosing between CSNBENC and CSNBENC1

CSNBENC and CSNBENC1 provide identical functions. When choosing which service to use, consider the following:

- **CSNBENC** requires the cleartext and ciphertext to reside in the caller's primary address space. Also, a program using CSNBENC adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBENC1** allows the cleartext and ciphertext to reside either in the caller's primary address space or in a data space. This can allow you to encipher more data with one call. However, a program using CSNBENC1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBENC1, *clear_text_id* and *cipher_text_id* are access list entry token (ALET) parameters of the data spaces containing the cleartext and ciphertext.

Format

```
CALL CSNBENC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             clear_text,
             initialization_vector,
             rule_array_count,
             rule_array,
             pad_character,
             chaining_vector,
             cipher_text )
```

```
CALL CSNBENC1(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier,
              text_length,
              clear_text,
              initialization_vector,
              rule_array_count,
              rule_array,
              pad_character,
              chaining_vector,
              cipher_text,
              clear_text_id,
              cipher_text_id )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

key_identifier

Direction: Input/Output	Type: String

A 64-byte string that is the internal key token containing the data-encrypting key, or the label of a CKDS record containing the data-encrypting key, to be used for encrypting the data. If the key token or key label contains a single-length key, single-DES encryption is performed. If the key token or key label contains a double-length or triple-length key, triple-DES encryption is performed.

text_length

Direction: Input/Output	Type: Integer

On entry, the length of the plaintext (*clear_text* parameter) you supply. The MAXLEN keyword in the options file, as provided by the installation, determines the upper limit of the length of the text, including any necessary padding. A zero value for the *text_length* parameter is not valid. If the returned enciphered text (*cipher_text* parameter) is a different length because of the addition of padding bytes, the value is updated to the length of the ciphertext.

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

The application program passes the length of the plaintext to the callable service. The callable service returns the length of the ciphertext to the application program.

clear_text

Direction: Input	Type: String

The text that is to be enciphered.

initialization_vector

Direction: Input	Type: String

The 8-byte supplied string for the cipher block chaining. The first 8 bytes (or less) block of the data is exclusive ORed with the ICV and then enciphered. The input block is enciphered and the next ICV is created. You must use the same ICV to decipher the data.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supply in the *rule_array* parameter. The value must be 1, 2, or 3.

rule_array

Direction: Input	Type: Character string

An array of 8-byte keywords providing the processing control information. The array is positional. See the keywords in Table 35. The first keyword in the array is the processing rule. You choose the processing rule you want the callable service to use for enciphering the data. The second keyword is the ICV selection keyword. The third keyword (or the second if the ICV selection keyword is allowed to default to INITIAL) is the encryption algorithm to use.

The service will fail if keyword DES is specified in the *rule_array* in a CDMF-only system. The service will likewise fail if the keyword CDMF is specified in the *rule_array* in a DES-only system.

Table 35. Keywords for the Encipher Rule Array Control Information

Keyword	Meaning
Processing Rule (required)	
CBC	Performs ANSI X3.102 cipher block chaining. The data must be a multiple of 8 bytes. An OCV is produced and placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.
CUSP	Performs ciphering that is compatible with IBM's CUSP and PCF products. The data can be of any length and does not need to be in multiples of 8 bytes. The ciphertext will be the same length as the plaintext. The CUSP/PCF OCV is placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the CUSP/PCF OCV from the previous call is used as the ICV for this call.
IPS	Performs ciphering that is compatible with IBM's IPS product. The data may be of any length and does not need to be in multiples of 8 bytes. The ciphertext will be the same length as the plaintext. The IPS OCV is placed in the *chaining_vector* parameter. If the ICV selection keyword CONTINUE is specified, the IPS OCV from the previous call is used as the ICV for this call.
X9.23	Performs cipher block chaining with 1 to 8 bytes of padding. This is compatible with the requirements in ANSI standard X9.23. If the data is not in exact multiples of 8 bytes, X9.23 pads the plaintext so that the ciphertext produced is an exact multiple of 8 bytes. The plaintext is padded to the next multiple 8 bytes, even if this adds 8 bytes. An OCV is produced.

Table 35. Keywords for the Encipher Rule Array Control Information (continued)

Keyword	Meaning
4700-PAD	Performs padding by extending the user's plaintext with the caller's specified pad character, followed by a one-byte binary count field that contains the total number of bytes added to the message. 4700-PAD pads the plaintext so that the ciphertext produced is an exact multiple of 8 bytes. An OCV is produced.
ICV Selection (optional)	
CONTINUE	This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the *chaining_vector* parameter points. CONTINUE is valid only for processing rules CBC, IPS, and CUSP.
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. INITIAL is the default value.
Encryption Algorithm (optional)	
CDMF	This specifies using the Commercial Data Masking Facility and ignoring the token marking. You cannot use double-length or triple-length keys with CDMF.
DES	This specifies using the data encryption standard and ignoring the token marking.
TOKEN	This specifies using the data encryption algorithm in the DATA key token. TOKEN is the default.

The following recommendations help the caller determine which encipher processing rule to use:

- If you are exchanging enciphered data with a specific implementation, for example, CUSP or ANSI X9.23, use that processing rule.
- If the ciphertext translate callable service is to be invoked on the enciphered data at an intermediate node, ensure that the ciphertext is a multiple of 8 bytes. Use CBC, X9.23, or 4700-PAD to prevent the creation of ciphertext that is not a multiple of 8 bytes and that cannot be processed by the ciphertext translate callable service.
- If the ciphertext length must be equal to the plaintext length and the plaintext length cannot be a multiple of 8 bytes, use either the IPS or CUSP processing rule.

Cipher Processing Rules describes the cipher processing rules in detail.

pad_character

Direction: Input	Type: Integer

An integer, 0 to 255, that is used as a padding character for the 4700-PAD process rule (*rule_array* parameter).

chaining_vector

Direction: Input/Output	Type: String

An 18-byte field that ICSF uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first 8 bytes in the 18-byte string.

The direction is output if the ICV selection keyword of the *rule_array* parameter is INITIAL.

The direction is input/output if the ICV selection keyword of the *rule_array* parameter is CONTINUE.

cipher_text

Direction: Output	Type: String

The enciphered text the callable service returns. The length of the ciphertext is returned in the *text_length* parameter. The *cipher_text* may be 8 bytes longer than the length of the *clear_text* field because of the padding that is required for some processing rules.

clear_text_id

Direction: Input	Type: Integer

For CSNBENC1 only, the ALET of the clear text to be enciphered.

cipher_text_id

Direction: Input	Type: Integer

For CSNBENC1 only, the ALET of the ciphertext that the application supplied.

Restrictions

The service will fail under the following conditions:

- If the keyword DES is specified in the *rule_array* parameter in a CDMF-only system
- If the keyword CDMF is specified in the *rule_array* parameter in a DES-only system
- If the key token contains double- or triple-length keys and triple-DES is not enabled.

Related Information

You **cannot** overlap the plaintext and ciphertext fields. For example:

```
pppppp
     cccccc  is incorrect.

cccccc
     pppppp  is incorrect.
ppppppcccccc is correct.

P represents the plaintext and c represents the ciphertext.
```

The method used to produce the OCV is the same with the CBC, 4700-PAD, and X9.23 processing rules. However, that method is different from the method used by the CUSP and IPS processing rules.

Cipher Processing Rules discusses the cipher processing rules.

The decipher callable services (CSNBDEC and CSNBDEC1) are described under "Decipher (CSNBDEC and CSNBDEC1)" on page 162.

Encode (CSNBECO)

Use the encode callable service (CSNBECO) to encipher an 8-byte string using a clear key. The callable service uses the electronic code book (ECB) mode of the DES. (This service is available only on a DES-capable system.)

Considerations

If you have only a clear key, you are *not* limited to using just the encode and decode callable services. You can pass your clear key to the clear key import service, and get back a token that will allow you to use the encipher and decipher callable services.

Format

```
CALL CSNBECO(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            clear_key,
            clear_text,
            cipher_text)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

clear_key

Direction: Input Type: String

The 8-byte clear key value that is used to encode the data.

clear_text

Direction: Input Type: String

The plaintext that is to be encoded. Specify 8 bytes of text.

cipher_text

Direction: Output Type: String

The 8-byte field where the ciphertext is returned by the callable service.

Restriction

You cannot use this service on a CDMF-only system.

Symmetric Key Decipher (CSNBSYD)

Use the symmetric key decipher callable service to decipher data in an address space or a data space using the cipher block chaining or electronic code book modes. The Advanced Encryption Standard (AES) is supported. AES encryption uses a 128-, 192-, or 256-bit key. The CBC and ECB modes are supported. Due to export regulations, AES encryption may not be available on your system.

This service supports both electronic code book (ECB) and cipher block chaining (CBC) modes. The CBC mode of operation uses an initial chaining vector (ICV) in its processing. The ICV is exclusive ORed with the first block of plaintext after the decryption step, and thereafter, each block of ciphertext is exclusive ORed with the next block of plaintext after decryption, and so on.

Cipher block chaining also produces a resulting chaining value called the output chaining vector (OCV). The application can pass the OCV as the ICV in the next encipher call. This results in record chaining.

The electronic code book mode does not use the initial chaining vector.

Format

```
CALL CSNBSYD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_length,
             key_identifier,
             key_parms_length,
             key_parms,
             block_size,
             initialization_vector_length,
             initialization_vector,
             chain_data_length,
             chain_data,
             cipher_text_length,
             cipher_text,
             clear_text_length,
             clear_text,
             optional_data_length,
             optional_data)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Ignored	Type: Integer

Reserved field.

exit_data

Direction: Ignored	Type: String

Reserved field.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value may be 1, 2, 3 or 4.

rule_array

Direction: Input Type: String

An array of 8-byte keywords providing the processing control information. The keywords must be in contiguous storage, left-justified and padded on the right with blanks.

Table 36. Symmetric Key Decipher Rule Array Keywords

Keyword	Meaning
Algorithm (required)	
AES	Specifies that the Advanced Encryption Standard (AES) algorithm is to be used. The block size is 16 bytes. The key length may be 16, 24, or 32 bytes. The *chain_data* field must be at least 32 bytes in length. The OCV is in the first 16 bytes in the *chain_data*.
Processing Rule (optional)	
CBC	Performs cipher block chaining. The text length must be a multiple of the block size for the specified algorithm. CBC is the default value.
ECB	Performs electronic code book encryption. The text length must be a multiple of the block size for the specified algorithm.
Key Rule (optional)	
KEY-CLR	This specifies that the key parameter contains a clear key value. KEY-CLR is the default value.
ICV Selection (optional)	
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. INITIAL is the default value.
CONTINUE	This specifies taking the initialization vector from the output chaining vector contained in the work area to which the *chain_data* parameter points. CONTINUE is valid only for processing rule CBC.

key_length

Direction: Input Type: Integer

The length of the key parameter. For clear keys, the length is in bytes and includes only the value of the key. The maximum size is 256 bytes.

key_identifier

Direction: Input Type: String

The cipher key. The parameter must be left justified.

key_parms_length

Direction: Ignored	Type: Integer

The length of the *key_parms* parameter. The maximum size is 256 bytes.

key_parms

Direction: Ignored	Type: String

This parameter contains key-related parameters specific to the encryption algorithm.

block_size

Direction: Input	Type: Integer

This parameter contains the processing size of the text block in bytes. This value will be algorithm specific. Be sure to specify the same block size as used to encipher the text.

initialization_vector_length

Direction: Input	Type: Integer

The length of the *initialization_vector* parameter. The length should be equal to the block length for the algorithm specified.

initialization_vector

Direction: Input	Type: String

This initialization chaining value for CBC encryption. You must use the same ICV that was used to encipher the data.

chain_data_length

Direction: Input/Output	Type: Integer

The length of the *chain_data* parameter. On output, the actual length of the chaining vector will be stored in the parameter.

chain_data

Direction: Input/Output	Type: String

This field is used as a system work area for the chaining vector. Your application program must not change the data in this string. The chaining vector holds the output chaining vector from the caller.

The direction is output if the ICV selection keyword is INITIAL.

The mapping of the *chain_data* depends on the algorithm specified. For AES, the *chain_data* field must be at least 32 bytes in length. The OCV is in the first 16 bytes in the *chain_data*.

cipher_text_length

Direction: Input	Type: Integer

The length of the cipher text. A zero value in the *clear_text_length* parameter is not valid. The length must be a multiple of the algorithm block size.

cipher_text

Direction: Input	Type: String

The text to be deciphered.

clear_text_length

Direction: Input/Output	Type: Integer

On input, this parameter specifies the size of the storage pointed to by the *clear_text* parameter. On output, this parameter has the actual length of the text stored in the *clear_text* parameter.

clear_text

Direction: Output	Type: String

The deciphered text the service returns.

optional_data_length

Direction: Ignored	Type: Integer

The length of the *optional_data* parameter.

optional_data

Direction: Ignored	Type: String

Optional data required by a specified algorithm.

Usage Notes

- No pre- or post-processing exits are enabled for this service.
- No SAF authorization check is made.
- The CCF master keys need not be loaded to use this service.
- The AES algorithm is implemented in the software.
- AES has the same availability restrictions as triple-DES.
- This service will fail if execution would cause destructive overlay of the *cipher_text* field.

Symmetric Key Encipher (CSNBSYE)

Use the symmetric key encipher callable service to encipher data in an address space or a data space using the cipher block chaining or electronic code book modes. The Advanced Encryption Standard (AES) is supported. AES encryption uses a 128-, 192-, or 256-bit key. The CBC and ECB modes are supported. Due to export regulations, AES encryption may not be available on your system.

This service supports both electronic code book (ECB) and cipher block chaining (CBC) modes. The CBC mode of operation uses an initial chaining vector (ICV) in its processing. The ICV is exclusive ORed with the first block of plaintext before the encryption step, and thereafter, the block of ciphertext just produced is exclusive ORed with the next block of plaintext, and so on. This disguises any pattern that may exist in the plaintext.

Cipher block chaining also produces a resulting chaining value called the output chaining vector (OCV). The application can pass the OCV as the ICV in the next encipher call. This results in record chaining.

The electronic code book mode does not use the initial chaining vector.

Format

```
CALL CSNBSYE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             key_length,
             key_identifier,
             key_parms_length,
             key_parms,
             block_size,
             initialization_vector_length,
             initialization_vector,
             chain_data_length,
             chain_data,
             clear_text_length,
             clear_text,
             cipher_text_length,
             cipher_text,
             optional_data_length,
             optional_data)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Ignored	Type: Integer

Reserved field.

exit_data

Direction: Ignored	Type: String

Reserved field.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value may be 1, 2, 3 or 4.

rule_array

Direction: Input	Type: String

An array of 8-byte keywords providing the processing control information. The keywords must be in contiguous storage, left-justified and padded on the right with blanks.

Table 37. Symmetric Key Encipher Rule Array Keywords

Keyword	Meaning
Algorithm (required)	
AES	Specifies that the Advanced Encryption Standard (AES) algorithm is to be used. The block size is 16 bytes. The key length may be 16, 24, or 32 bytes. The *chain_data* field must be at least 32 bytes in length. The OCV is in the first 16 bytes in the *chain_data*.
Processing Rule (optional)	
CBC	Performs cipher block chaining. The text length must be a multiple of the block size for the specified algorithm. CBC is the default value.
ECB	Performs electronic code book encryption. The text length must be a multiple of the block size for the specified algorithm.
Key Rule (optional)	
KEY-CLR	This specifies that the key parameter contains a clear key value. KEY-CLR is the default.
ICV Selection (optional)	
INITIAL	This specifies taking the initialization vector from the *initialization_vector* parameter. INITIAL is the default value.

Table 37. Symmetric Key Encipher Rule Array Keywords (continued)

Keyword	Meaning
CONTINUE	This specifies taking the initialization vector from the output chaining vector contained in the work area to which the *chain_data* parameter points. CONTINUE is valid only for processing rule CBC.

key_length

Direction: Input	Type: Integer

The length of the key parameter. For clear keys, the length is in bytes and includes only the value of the key.

key_identifier

Direction: Input	Type: String

The cipher key. The parameter must be left justified.

key_parms_length

Direction: Ignored	Type: Integer

The length of the *key_parms* parameter.

key_parms

Direction: Ignored	Type: String

This parameter contains key-related parameters specific to the encryption algorithm.

block_size

Direction: Input	Type: Integer

This parameter contains the processing size of the text block in bytes. This value will be algorithm specific.

initialization_vector_length

Direction: Input	Type: Integer

The length of the *initialization_vector* parameter. The length should be equal to the block length for the algorithm specified.

initialization_vector

Direction: Input	Type: String

This initialization chaining value for CBC encryption. You must use the same ICV to decipher the data.

chain_data_length

Direction: Input/Output	Type: Integer

The length of the *chain_data* parameter. On output, the actual length of the chaining vector will be stored in the parameter.

chain_data

Direction: Input/Output	Type: String

This field is used as a system work area for the chaining vector. Your application program must not change the data in this string. The chaining vector holds the output chaining vector from the caller.

The direction is output if the ICV selection keyword is INITIAL.

The mapping of the *chain_data* depends on the algorithm specified. For AES, the *chain_data* field must be at least 32 bytes in length. The OCV is in the first 16 bytes in the *chain_data*.

clear_text_length

Direction: Input	Type: Integer

The length of the clear text. A zero value in the *clear_text_length* parameter is not valid. The length must be a multiple of the algorithm block size.

clear_text

Direction: Input	Type: String

The text to be enciphered.

cipher_text_length

Direction: Input/Output	Type: Integer

On input, this parameter specifies the size of the storage pointed to by the *cipher_text* parameter. On output, this parameter has the actual length of the text stored in the buffer addressed by the *cipher_text* parameter.

cipher_text

Direction: Output	Type: String

The enciphered text the service returns.

optional_data_length

Direction: Ignored	Type: Integer

The length of the *optional_data* parameter.

optional_data

Direction: Ignored	Type: String

Optional data required by a specified algorithm.

Usage Notes

- No pre- or post-processing exits are enabled for this service.
- No SAF authorization check is made.
- The CCF master keys need not be loaded to use this service.
- The AES algorithm is implemented in the software.
- AES has the same availability restrictions as triple-DES.
- This service will fail if execution would cause destructive overlay of the *clear_text* field.

Chapter 6. Verifying Data Integrity and Authenticating Messages

ICSF provides several methods to verify the integrity of transmitted messages and stored data:

- Message authentication code (MAC)
- Hash functions, including modification detection code (MDC) processing and one-way hash generation

Note: You can also use digital signatures (see Chapter 8, “Using Digital Signatures” on page 265) to authenticate messages.

The choice of callable service depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender as well as the integrity of the data, and both the sender and receiver can share a secret key, consider message authentication code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions, such as the modification detection code process.

The callable services are described in the following topics:

- “MAC Generate (CSNBMGN and CSNBMGN1)” on page 191
- “MAC Verify (CSNBMVR and CSNBMVR1)” on page 195
- “MDC Generate (CSNBMDG and CSNBMDG1)” on page 200
- “One-Way Hash Generate (CSNBOWH and CSNBOWH1)” on page 204

How MACs are Used

When a message is sent, an application program can generate an authentication code for it using the MAC generation callable service. ICSF supports the ANSI X9.9-1 basic procedure and both the ANSI X9.19 basic procedure and optional double key MAC procedure. The service computes the text of the message authentication code using the algorithm and a key. The ANSI X9.9-1 or ANSI X9.19 basic procedures accept either a single-length MAC generation (MAC) key or a data-encrypting (DATA) key, and the message text. The ANSI X9.19 optional double key MAC procedure accepts a double-length MAC key and the message text. The message text may be in clear or encrypted form. The originator of the message sends the MAC with the message text.

When the receiver gets the message, an application program calls the *MAC verification callable service*. The callable service generates a MAC using the same algorithm as the sender and either the single-length (MACVER) or double-length (DATAMV) MAC verification key, the single-length (MAC) or double-length (DATAM) MAC generation key, or DATA key, and the message text. The MACVER callable service compares the MAC it generates with the one sent with the message and issues a return code that indicates whether the MACs match. If the return code indicates that the MACs match, the receiver can accept the message as genuine and unaltered. If the return code indicates that the MACs do not match, the receiver can assume that the message is either bogus or has been altered. The newly computed MAC is not revealed outside the cryptographic feature.

In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

Secure use of the MAC generation and MAC verification services requires the use of MAC and MACVER keys in these services, respectively. To accomplish this, the originator of the message generates a MAC/MACVER key pair, uses the MAC key in the MAC generation service, and exports the MACVER key to the receiver. The originator of the message enforces key separation on the link by encrypting the MACVER key under a transport key that is not an NOCV key before exporting the key to the receiver. With this type of key separation enforced, the receiver can only receive a MACVER key and can use only this key in the MAC verification service. This ensures that the receiver cannot alter the message and produce a valid MAC with the altered message. These security features are not present if DATA keys are used in the MAC generation service, or if DATA or MAC keys are used in the MAC verification service.

By using MACs, you get the following benefits:

- **For data transmitted over a network,** you can validate the authenticity of the message as well as ensure that the data has not been altered during transmission. For example, an active eavesdropper can tap into a transmission line, and interject bogus messages or alter sensitive data being transmitted. If the data is accompanied by a MAC, the recipient can use a callable service to detect whether the data has been altered. Since both the sender and receiver share a secret key, the receiver can use a callable service that calculates a MAC on the received message and compares it to the MAC transmitted with the message. If the comparison is equal, the message may be accepted as unaltered. Furthermore, since the shared key is secret, when a MAC is verified it can be assumed that the sender was, in fact, the other person who knew the secret key.
- **For data stored on tape or DASD,** you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. For example, someone might be able to bypass access controls. Such an access might escape the notice of auditors. However, if a MAC is stored with the data, and verified when the data is read, you can detect alterations to the data.

How Hashing Functions Are Used

Hashing functions include the MDC and one-way hash. You need to hash text before submitting it to digital signature services (see Chapter 8, “Using Digital Signatures” on page 265).

How MDCs Are Used

When a message is sent, an application program can generate a modification detection code for it using the *MDC generation callable service*. The service computes the modification detection code, a 128-bit value, using a one-way cryptographic function and the message text (which itself may be in clear or encrypted form). The originator of the message ensures that the MDC is transmitted with integrity to the intended receiver of the message. For example, the MDC could be published in a reliable source of public information.

When the receiver gets the message, an application program calls the *MDC callable service*. The callable service generates an MDC by using the same one-way cryptographic function and the message text. The application program can compare the new MDC with the one generated by the originator of the message. If the MDCs match, the receiver knows that the message was not altered.

In a similar manner, MDCs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

By using MDCs, you get the following benefits:

- **For data transmitted over a network between locations that do not share a secret key,** you can ensure that the data has not been altered during transmission. It is easy to compute an MDC for specific data, yet hard to find data that will result in a given MDC. In effect, the problem of ensuring the integrity of a large file is reduced to ensuring the integrity of a 128-bit value.
- **For data stored on tape or DASD,** you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. Once an MDC has been established for a file, the MDC generation callable service can be run at any later time on the file. The resulting MDC can be compared with the stored MDC to detect deliberate or inadvertent modification.

SHA-1 is a FIPS standard required for DSS. MD5 is a hashing algorithm used to derive Message Digests in Digital Signature applications.

MAC Generate (CSNBMGN and CSNBMGN1)

Use the MAC generate callable service to generate a 4-, 6-, or 8-byte message authentication code (MAC) for an application-supplied text string. You can specify that the callable service uses either the ANSI X9.9-1 procedure or the ANSI X9.19 optional double key MAC procedure to compute the MAC. For the ANSI X9.9-1 procedure you identify either a MAC generate key or a DATA key, and the message text. For the ANSI X9.19 optional double key MAC procedure, you identify a double-length MAC key and the message text.

The MAC generate callable service also supports the padding rules specified in the EMV Specification.

ICSF routes the MAC Generate request to a PCI Cryptographic Coprocessor if the control vector in the supplied key identifier cannot be processed on the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails. The request must meet the following restrictions:

- The MAC Process Rule is X9.19OPT or EMVMACD
- The MAC key is a valid double-length MAC generate key
- The segmenting control keyword is ONLY
- The *text_length* is not greater than 4096 including padding

Choosing Between CSNBMGN and CSNBMGN1

CSNBMGN and CSNBMGN1 provide identical functions. When choosing which service to use, consider the following:

- **CSNBMGN** requires the application-supplied text to reside in the caller's primary address space. Also, a program using CSNBMGN adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBMGN1** allows the application-supplied text to reside either in the caller's primary address space or in a data space. This can allow you to process more data with one call. However, a program using CSNBMGN1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBMGN1, *text_id_in* is an access list entry token (ALET) parameter of the data space containing the application-supplied text.

Format

```
CALL CSNBMGN(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            key_identifier,
            text_length,
            text,
            rule_array_count,
            rule_array,
            chaining_vector,
            mac )
```

```
CALL CSNBMGN1(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             mac,
             text_id_in )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

key_identifier

Direction: Input/Output Type: String

The 64-byte key label or internal key token that identifies a single-length or double-length MAC generate key or a single-length DATA key. The type of key depends on the MAC process rule in the *rule_array* parameter.

text_length

Direction: Input Type: Integer

The length of the text you supply in the *text* parameter. The MAXLEN keyword in the options file determines the maximum length of the text. (See *z/OS ICSF System Programmer's Guide* for a description of the MAXLEN keyword in the options file.) If the *text_length* is not a multiple of 8 bytes and if the ONLY or LAST keyword of the *rule_array* parameter is called, the text is padded with binary zeros.

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

text

Direction: Input Type: String

The application-supplied text for which the MAC is generated.

rule_array_count

Direction: Input Type: Integer

The number of keywords specified in the *rule_array* parameter. The value can be 0, 1, 2, or 3.

rule_array

Direction: Input Type: Character string

Zero to three keywords that provide control information to the callable service. The keywords are shown in Table 38 on page 194. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. For example,

```
'X9.9-1  MIDDLE  MACLEN4 '
```

The order of the *rule_array* keywords is not fixed.

You can specify one of the MAC processing rules and then choose one of the segmenting control keywords and one of the MAC length keywords.

Table 38. Keywords for MAC generate Control Information

Keyword	Meaning
MAC Process Rules (optional)	
EMVMAC	EMV padding rule with a single-length MAC key. The *key_identifier* parameter must identify a single-length MAC or a single-length DATA key. The text is always padded with 1 to 8 bytes so that the resulting text length is a multiple of 8 bytes. The first pad character is X'80'. The remaining 0 to 7 pad characters are X'00'.
EMVMACD	EMV padding rule with a double-length MAC key. The *key_identifier* parameter must identify a double-length MAC key. The padding rules are the same as for EMVMAC.
X9.19OPT	ANSI X9.19 optional double key MAC procedure. The *key_identifier* parameter must identify a double-length MAC key. The padding rules are the same as for X9.9-1.
X9.9-1	ANSI X9.9-1 and X9.19 basic procedure. The *key_identifier* parameter must identify a single-length MAC or a single-length DATA key. X9.9-1 causes the MAC to be computed from all of the data. The text is padded only if the text length is not a multiple of 8 bytes. If padding is required, the pad character X'00' is used. This is the default value.
Segmenting Control (optional)	
FIRST	First call, this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; segmenting is not employed by the application program. This is the default value.
MAC Length and Presentation (optional)	
HEX-8	Generates a 4-byte MAC value and presents it as 8 hexadecimal characters.
HEX-9	Generates a 4-byte MAC value and presents it as 2 groups of 4 hexadecimal characters with a space between the groups.
MACLEN4	Generates a 4-byte MAC value. This is the default value.
MACLEN6	Generates a 6-byte MAC value.
MACLEN8	Generates an 8-byte MAC value.

chaining_vector

Direction: Input/Output Type: String

An 18-byte string that ICSF uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac

Direction: Output Type: String

The 8-byte or 9-byte field in which the callable service returns the MAC value if the segmenting rule is ONLY or LAST. Allocate an 8-byte field for MAC values of 4 bytes, 6 bytes, 8 bytes, or HEX-8. Allocate a 9-byte MAC field if you specify HEX-9 in the *rule_array* parameter.

text_id_in

Direction: Input	Type: Integer

For CSNBMGN1 only, the ALET of the text for which the MAC is generated.

Usage Notes

To use a DATA key, the NOCV-enablement keys must be present in the CKDS. Using a DATA key instead of a MAC generate key in this service substantially increases the path length for generating the MAC.

To calculate a MAC in one call, specify the ONLY keyword for segmenting control for the *rule_array* parameter. For two or more calls, specify the FIRST keyword for the first input block, the MIDDLE keyword for intermediate blocks (if any), and the LAST keyword for the last block.

For a given text string, the resulting MAC is the same whether the text is segmented or not.

Related Information

For more information about MAC processing rules and segmenting control, refer to *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*.

The MAC verification callable service is described in “MAC Verify (CSNBMVR and CSNBMVR1)”.

MAC Verify (CSNBMVR and CSNBMVR1)

Use the MAC verify callable service to verify a 4-, 6-, or 8-byte message authentication code (MAC) for an application-supplied text string. You can specify that the callable service uses either the ANSI X9.9-1 procedure or the ANSI X9.19 optional double key MAC procedure to compute the MAC. For the ANSI X9.9-1 procedure you identify either a MAC verify key, a MAC generation key, or a DATA key, and the message text. For the ANSI X9.19 optional double key MAC procedure, you identify either a double-length MAC verify key or a double-length MAC generation key and the message text. The cryptographic feature compares the generated MAC with the one sent with the message. A return code indicates whether the MACs are the same. If the MACs are the same, the receiver knows the message was not altered. The generated MAC never appears in storage is not revealed outside the cryptographic feature.

The MAC verify callable service also supports the padding rules specified in the EMV Specification.

ICSF routes the MAC Verify request to a PCI Cryptographic Coprocessor if the control vector in the supplied key identifier cannot be processed on the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails. The request must meet the following restrictions:

- The MAC Process Rule is X9.19OPT or EMVMACD

- The MAC key is a valid double-length MAC generation or MAC verify key
- The segmenting control keyword is ONLY
- The *text_length* is not greater than 4096 including padding

Choosing Between CSNBMVR and CSNBMVR1

CSNBMVR and CSNBMVR1 provide identical functions. When choosing which service to use, consider the following:

- **CSNBMVR** requires the application-supplied text to reside in the caller's primary address space. Also, a program using CSNBMVR adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBMVR1** allows the application-supplied text to reside either in the caller's primary address space or in a data space. This can allow you to verify more data with one call. However, a program using CSNBMVR1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface, and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBMVR1, *text_id_in* is an access list entry token (ALET) parameter of the data space containing the application-supplied text.

Format

```
CALL CSNBMVR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             key_identifier,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             mac )
```

```
CALL CSNBMVR1(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              key_identifier,
              text_length,
              text,
              rule_array_count,
              rule_array,
              chaining_vector,
              mac,
              text_id_in )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

key_identifier

Direction: Input/Output

The 64-byte key label or internal key token that identifies a single-length or double-length MAC verify key, a single-length or double-length MAC generation key or a single-length DATA key. The type of key depends on the MAC process rule in the *rule_array* parameter.

text_length

Direction: Input	Type: Integer

The length of the clear text you supply in the *text* parameter. The MAXLEN keyword in the options file determines the maximum length of the text. (See the *z/OS ICSF System Programmer’s Guide* for a description of the MAXLEN keyword in the options file.) If the *text_length* parameter is not a multiple of 8 bytes and if the ONLY or LAST keyword of the *rule_array* parameter is called, the text is padded with binary zeros.

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

text

Direction: Input	Type: String

The application-supplied text for which the MAC is verified.

rule_array_count

Direction: Input	Type: Integer

The number of keywords specified in the *rule_array* parameter. The value can be 0, 1, 2, or 3.

rule_array

Direction: Input	Type: Character string

Zero to three keywords that provide control information to the callable service. The keywords are shown in Table 39. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks. For example,

```
'X9.9-1  MIDDLE  MACLEN4 '
```

The order of the *rule_array* keywords is not fixed.

You can specify one of the MAC processing rules and then choose one of the segmenting control keywords and one of the MAC length keywords.

Table 39. Keywords for MAC verify Control Information

Keyword	Meaning
MAC Process Rules (optional)	
EMVMAC	EMV padding rule with a single-length MAC key. The *key_identifier* parameter must identify a single-length MAC, MACVER, or DATA key. The text is always padded with 1 to 8 bytes so that the resulting text length is a multiple of 8 bytes. The first pad character is X'80'. The remaining 0 to 7 pad characters are X'00'.
EMVMACD	EMV padding rule with a double-length MAC key. The *key_identifier* parameter must identify a double-length MAC or MACVER key. The padding rules are the same as for EMVMAC.
X9.9-1	ANSI X9.9-1 and X9.19 basic procedure. The *key_identifier* parameter must identify a single-length MAC, MACVER, or DATA key. X9.9-1 causes the MAC to be computed from all of the data. The text is padded only if the text length is not a multiple of 8 bytes. If padding is required, the pad character X'00' is used. This is the default value.
X9.19OPT	ANSI X9.19 optional double-length MAC procedure. The *key_identifier* parameter must identify a double-length MAC or MACVER key. The padding rules are the same as for X9.9-1.
Segmenting Control (optional)	
FIRST	First call; this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; the application program does not employ segmenting. This is the default value.
MAC Length and Presentation (optional)	

Table 39. Keywords for MAC verify Control Information (continued)

Keyword	Meaning
HEX-8	Verifies a 4-byte MAC value that is represented as 8 hexadecimal characters.
HEX-9	Verifies a 4-byte MAC value that is represented as 2 groups of 4 hexadecimal characters with a space character between the groups.
MACLEN4	Verifies a 4-byte MAC value. This is the default value.
MACLEN6	Verifies a 6-byte MAC value.
MACLEN8	Verifies an 8-byte MAC value.

chaining_vector

Direction: Input/Output	Type: String

An 18-byte string that ICSF uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter to binary zeros.

mac

Direction: Input	Type: String

The 8- or 9-byte field that contains the MAC value you want to verify. The value in the field must be left-justified and padded with zeros. If you specified the HEX-9 keyword in the *rule_array* parameter, the input MAC is 9 bytes.

text_id_in

Direction: Input	Type: Integer

For CSNBMVR1 only, the ALET of the text for which the MAC is to be verified.

Usage Notes

To verify a MAC in one call, specify the ONLY keyword on the segmenting rule keyword for the *rule_array* parameter. For two or more calls, specify the FIRST keyword for the first input block, MIDDLE for intermediate blocks (if any), and LAST for the last block.

For a given text string, the MAC resulting from the verification process is the same regardless of how the text is segmented, or how it was segmented when the original MAC was generated.

To use a MAC generation key or a DATA key, the NOCV enablement keys must be present in the CKDS. Using either a MAC generation key or a DATA key instead of a MAC verify key in this service substantially increases the path length for verifying the MAC.

Related Information

For more information about MAC processing rules and segmenting control, refer to *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*.

The MAC generation callable service is described in “MAC Generate (CSNBMGN and CSNBMGN1)” on page 191.

MDC Generate (CSNBMDG and CSNBMDG1)

A modification detection code (MDC) can be used to provide a form of support for data integrity.

Use the MDC generate callable service to generate a 128-bit modification detection code (MDC) for an application-supplied text string.

The returned MDC value should be securely stored and/or sent to another user. To validate the integrity of the text string at a later time, the MDC generate callable service is again used to generate a 128-bit MDC. The new MDC value is compared with the original MDC value. If the values are equal, the text is accepted as unchanged.

Choosing Between CSNBMDG and CSNBMDG1

CSNBMDG and CSNBMDG1 provide identical functions. When choosing which service to use, consider the following:

- **CSNBMDG** requires the application-supplied text to reside in the caller’s primary address space. Also, a program using CSNBMDG adheres to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface.
- **CSNBMDG1** allows the application-supplied text to reside either in the caller’s primary address space or in a data space. This can allow you to process more data with one call. However, a program using CSNBMDG1 does not adhere to the IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface and may need to be modified before it can run with other cryptographic products that follow this programming interface.

  For CSNBMDG1, *text_id_in* parameter specifies the access list entry token (ALET) for the data space containing the application-supplied text.

Format

```
CALL CSNBMDG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             text_length,
             text,
             rule_array_count,
             rule_array,
             chaining_vector,
             mdc )
```

```
CALL CSNBMDG1(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              text_length,
              text,
              rule_array_count,
              rule_array,
              chaining_vector,
              mdc,
              text_id_in )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” on page 335 lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” on page 335 lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

text_length

Direction: Input Type: Integer

The length of the text you supply in the *text* parameter.

The MAXLEN keyword in the options file determines the maximum length of the text for this call. (See *z/OS ICSF System Programmer’s Guide* for a description of the MAXLEN keyword in the options file.)

Note: Beginning in z/OS V1 R2, the MAXLEN value may still be specified in the options data set, but only the maximum value limit will be enforced (2147483647).

Additional restrictions on length of the text depend on whether padding of the text is requested, and on the segmenting control used.

- When padding is requested (by specifying a process rule of PADMDC-2 or PADMDC-4 in the *rule_array* parameter), a text length of 0 is valid for any segment control specified in the *rule_array* parameter (FIRST, MIDDLE, LAST, or ONLY). When LAST or ONLY is specified, the supplied text will be padded with X'FF's and a padding count in the last byte to bring the total text length to the next multiple of 8 that is greater than or equal to 16,
- When no padding is requested (by specifying a process rule of MDC-2 or MDC-4), the total length of the text provided (over a single or segmented calls) must be at least 16 bytes, and a multiple of 8.

  For segmented calls with no padding, text length of 0 is valid on any of the calls provided the total length over the segmented calls is at least 16 and a multiple of 8.

  For a single call (that is, segment control is ONLY) with no padding, the length the text provided must be at least 16, and a multiple of 8.

text

Direction: Input	Type: String

The application-supplied text for which the MDC is generated.

rule_array_count

Direction: Input	Type: Integer

The number of keywords specified in the *rule_array* parameter. This value must be 2.

rule_array

Direction: Input	Type: Character string

The two keywords that provide control information to the callable service are shown in Table 40. The two keywords must be in 16 bytes of contiguous storage with each of the two keywords left-justified in its own 8-byte location and padded on the right with blanks. For example,

```
'MDC-2   FIRST   '
```

Choose one of the MDC process rule control keywords and one of the segmenting control keywords from the following table.

Table 40. Keywords for MDC Generate Control Information

Keyword	Meaning
MDC Process Rules (required)	
MDC-2	MDC-2 specifies two encipherments per 8 bytes of input text and no padding of the input text.

Table 40. Keywords for MDC Generate Control Information (continued)

Keyword	Meaning
MDC-4	MDC-4 specifies four encipherments per 8 bytes of input text and no padding of the input text.
PADMDC-2	PADMDC-2 specifies two encipherments per 8 bytes of input text and padding of the input text. When the segment rule specifies ONLY or LAST, the input text is padded with X'FF's and a padding count in the last byte to bring the total text length to the next even multiple of 8 that is greater than, or equal to, 16.
PADMDC-4	PADMDC-4 specifies four encipherments per 8 bytes of input text and padding of the input text. When the segment rule specifies ONLY or LAST, the input text is padded with X'FF's and a padding count in the last byte to bring the total text length to the next even multiple of 8 that is greater than, or equal to, 16.
Segmenting Control (required)	
FIRST	First call; this is the first segment of data from the application program.
LAST	Last call; this is the last data segment.
MIDDLE	Middle call; this is an intermediate data segment.
ONLY	Only call; segmenting is not employed by the application program.

chaining_vector

Direction: Input/Output Type: String

An 18-byte string that ICSF uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mdc

Direction: Input/Output Type: String

A 16-byte field in which the callable service returns the MDC value when the segmenting rule is ONLY or LAST. When the segmenting rule is FIRST or MIDDLE, the value returned in this field is an intermediate MDC value that will be used as input for a subsequent call and must not be changed by the application program.

text_id_in

Direction: Input Type: Integer

For CSNBMDG1 only, the ALET for the data space containing the text for which the MDC is to be generated.

Usage Notes

To calculate an MDC in one call, specify the ONLY keyword for segmenting control in the *rule_array* parameter. For more than one call, specify the FIRST keyword for the first input block, the MIDDLE keyword for any intermediate blocks, and the LAST keyword for the last block. For a given text string, the resulting MDC is the same whether the text is segmented or not.

The two versions of MDC calculation (with two or four encipherments per 8 bytes of input text) allow the caller to trade a performance improvement for a decrease in security. Since 2 encipherments create results different from the results of 4 encipherments, ensure that you use the same number of encipherments to verify the MDC value.

Related Information

The MAC generation service, using a publicly known key, can be used similarly to the MDC generate service, while providing better performance. The cryptographic work factor to break a MAC generated with a publicly known key, although much less than that for MDC generate, may be enough for some applications.

One-Way Hash Generate (CSNBOWH and CSNBOWH1)

Use the one-way hash generate callable service to generate a one-way hash on specified text. This service supports the following methods:

- MD5
- SHA-1
- RIPEMD-160

Note: MDC-2, MDC-4, PADMDC-2 and PADMDC-4 are supported through the MDC generation service (see "MDC Generate (CSNBMDG and CSNBMDG1)" on page 200).

Format

```
CALL CSNBOWH(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             text_length,
             text,
             chaining_vector_length,
             chaining_vector,
             hash_length,
             hash)
```

```
CALL CSNBOWH1(
              return_code,
              reason_code,
              exit_data_length,
              exit_data,
              rule_array_count,
              rule_array,
              text_length,
              text,
              chaining_vector_length,
              chaining_vector,
              hash_length,
              hash,
              text_id_in)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be 1 or 2.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service are listed in Table 41. The optional chaining flag keyword indicates whether calls to this service are chained together logically to overcome buffer size limitations. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

Table 41. Keywords for One-Way Hash Generate Rule Array Control Information

Keyword	Meaning
Hash Method (required)	
MD5	Hash algorithm is MD5 algorithm. Use this hash method for PKCS-1.0 and PKCS-1.1. Length of hash generated is 16 bytes.
RPMD-160	Hash algorithm is RIPEMD-160. Length of hash generated is 20 bytes.
SHA-1	Hash algorithm is SHA-1 algorithm. Use this hash method for DSS. Length of hash generated is 20 bytes.
Chaining Flag (optional)	
FIRST	Specifies this is the first call in a series of chained calls. Intermediate results are stored in the *hash* field.
LAST	Specifies this is the last call in a series of chained calls.
MIDDLE	Specifies this is a middle call in a series of chained calls. Intermediate results are stored in the *hash* field.
ONLY	Specifies this is the only call and the call is not chained. This is the default.

text_length

Direction: Input Type: Integer

The length of the *text* parameter in bytes.

Note: If you specify the FIRST or MIDDLE keyword, then the text length must be a multiple of the blocksize of the hash method. For MD5 and SHA-1, this is a multiple of 64 bytes.

For ONLY and LAST, this service performs the required padding according to the algorithm specified.

text

Direction: Input Type: String

The application-supplied text on which this service performs the hash.

chaining_vector_length

Direction: Input Type: Integer

The byte length of the *chaining_vector* parameter. This must be 128 bytes.

chaining_vector

Direction: Input/Output Type: String

This field is a 128-byte work area. Your application must not change the data in this string. The chaining vector permits chaining data from one call to another.

hash_length

Direction: Input Type: Integer

The length of the supplied *hash* field in bytes.

Note: For SHA-1 and RPMD-160 this must be at least 20 bytes; for MD5 this must be at least 16 bytes.

hash

Direction: Input/Output Type: String

This field contains the hash, left-justified. The processing of the rest of the field depends on the implementation. If you specify the FIRST or MIDDLE keyword, this field contains the intermediate hash value. Your application must not change the data in this field between the sequence of FIRST, MIDDLE, and LAST calls for a specific message.

text_id_in

Direction: Input Type: Integer

For CSNBOWH1 only, the ALET for the data space containing the text for which to generate the hash.

Usage Note

Although MD5 and SHA-1 allow it, bit length text is not supported for any hashing method.

Chapter 7. Financial Services

The process of validating personal identities in a financial transaction system is called *personal authentication*. The personal identification number (PIN) is the basis for verifying the identity of a customer across financial industry networks. ICSF provides callable services to translate, verify, and generate PINs. You can use the callable services to prevent unauthorized disclosures when organizations handle PINs.

The following callable services are described in the following topics:
- "Clear PIN Encrypt (CSNBCPE)" on page 214
- "Clear PIN Generate (CSNBPGN)" on page 217
- "Clear PIN Generate Alternate (CSNBCPA)" on page 220
- "Encrypted PIN Generate (CSNBEPG)" on page 226
- "Encrypted PIN Translate (CSNBPTR)" on page 230
- "Encrypted PIN Verify (CSNBPVR)" on page 236
- "Secure Messaging for Keys (CSNBSKY)" on page 242
- "Secure Messaging for PINs (CSNBSPN)" on page 245
- "SET Block Compose (CSNDSBC)" on page 249
- "SET Block Decompose (CSNDSBD)" on page 253
- "VISA CVV Service Generate (CSNBCSG)" on page 258
- "VISA CVV Service Verify (CSNBCSV)" on page 261

How Personal Identification Numbers (PINs) are Used

Many people are familiar with PINs, which allow them to use an automated teller machine (ATM). From the system point of view, PINs are used primarily in financial networks to authenticate users — typically, a user is assigned a PIN, and enters the PIN at automated teller machines (ATMs) to gain access to his or her accounts. It is extremely important that the PIN be kept private, so that no one other than the account owner can use it. ICSF allows your applications to generate PINs, to verify supplied PINs, and to translate PINs from one format to another.

How VISA Card Verification Values Are Used

The Visa International Service Association (VISA) and MasterCard International, Incorporated have specified a cryptographic method to calculate a value that relates to the personal account number (PAN), the card expiration date, and the service code. The VISA card-verification value (CVV) and the MasterCard card-verification code (CVC) can be encoded on either track 1 or track 2 of a magnetic striped card and are used to detect forged cards. Because most online transactions use track-2, the ICSF callable services generate and verify the CVV[4] by the track-2 method.

The VISA CVV service generate callable service calculates a 1- to 5-byte value through the DES-encryption of the PAN, the card expiration date, and the service code using two data-encrypting keys or two MAC keys. The VISA CVV service verify callable service calculates the CVV by the same method, compares it to the CVV supplied by the application (which reads the credit card's magnetic stripe) in the *CVV_value*, and issues a return code that indicates whether the card is authentic.

[4] The VISA CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.

Translating Data and PINs in Networks

More and more data is being transmitted across networks where, for various reasons, the keys used on one network cannot be used on another network. Encrypted data and PINs that are transmitted across these boundaries must be “translated” securely from encryption under one key to encryption under another key. For example, a traveler visiting a foreign city might wish to use an ATM to access an account at home. The PIN entered at the ATM might need to be encrypted at the ATM and sent over one or more financial networks to the traveler’s home bank. At the home bank, the PIN must be verified before access is allowed. On intermediate systems (between networks), applications can use the Encrypted PIN translate callable service to re-encrypt a PIN block from one key to another. Running on ICSF, such applications can ensure that PINs never appear in the clear and that the PIN-encrypting keys are isolated on their own networks.

PIN Callable Services

You use the PIN callable services to generate, verify, and translate PINs. This section discusses the PIN callable services, as well as the various PIN algorithms and PIN block formats supported by ICSF. It also explains the use of PIN-encrypting keys.

Generating a PIN

To generate personal identification numbers, call the Clear PIN Generate or Encrypted PIN Generate callable service. Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the Clear PIN generate callable service generates a clear PIN and a PIN verification value, or offset. The Clear PIN Generate callable service can only execute in special secure mode. For a description of this mode, see “Special Secure Mode” on page 9. Using a PIN generation algorithm, data used in the algorithm, the PIN generation key, and an outbound PIN encrypting key, the encrypted PIN generate callable service generates and formats a PIN and encrypts the PIN block.

Encrypting a PIN

To format a PIN into a supported PIN block format and encrypt the PIN block, call the Clear PIN encrypt callable service.

Generating a PIN Validation Value from an Encrypted PIN Block

To generate a clear VISA PIN validation value (PVV) from an encrypted PIN block, call the *clear PIN generate alternate* callable service. The PIN block can be encrypted under an input PIN-encrypting key (IPINENC) or an output PIN encrypting key (OPINENC). Using an IPINENC key requires that NOCV keys are enabled in the CKDS.

Verifying a PIN

To verify a supplied PIN, call the *Encrypted PIN verify* callable service. You supply the enciphered PIN, the PIN-encrypting key that enciphers the PIN, and other data. You must also specify the PIN verification key and PIN verification algorithm. The callable service generates a verification PIN. The service compares the two personal identification numbers and if they are the same, it verifies the supplied PIN.

Translating a PIN

To translate a PIN block format from one PIN-encrypting key to another or from one PIN block format to another, call the *Encrypted PIN translate* callable service. You must identify the input PIN-encrypting key that originally enciphered the PIN. You also need to specify the output PIN-encrypting key that you want the callable service to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.

Algorithms for Generating and Verifying a PIN

ICSF supports the following algorithms for generating and verifying personal identification numbers:

- IBM 3624 institution-assigned PIN
- IBM 3624 customer-selected PIN (through a PIN offset)
- IBM German Bank Pool PIN (verify through an institution key)
- IBM German Bank Pool PIN (verify through a pool key and a PIN offset). This algorithm is supported when the service using the PIN is processed on the Cryptographic Coprocessor Feature.
- VISA PIN through a VISA PIN validation value
- Interbank PIN

The algorithms are discussed in detail in PIN Formats and Algorithms.

Using PINs on Different Systems

ICSF allows you to translate different PIN block formats, which lets you use personal identification numbers on different systems. ICSF supports the following formats:

- IBM 3624
- IBM 3621 (same as IBM 5906)
- IBM 4704 encrypting PINPAD format
- ISO 0 (same as ANSI 9.8, VISA 1, and ECI 1)
- ISO 1 (same as ECI 4)
- ISO 2
- VISA 2
- VISA 3
- VISA 4
- ECI 2
- ECI 3

The formats are discussed in “PIN Formats and Algorithms” on page 419.

PIN-Encrypting Keys

A unique master key variant enciphers each type of key. For further key separation, an installation can choose to have each PIN block format enciphered under a different PIN-encrypting key. The PIN-encrypting keys can have a 16-byte PIN block variant constant exclusive ORed on them before they are used to translate or verify PIN blocks. This is specified in the format control field in the Encrypted PIN translate and Encrypted PIN verify callable services.

You should only use PIN block variant constants when you are communicating with another host processor with the Integrated Cryptographic Service Facility.

For more information about PIN-encrypting keys, see *z/OS ICSF Administrator’s Guide*.

The PIN Profile

The PIN profile consists of the following:

- PIN block format
- Format control
- Pad digit
- Current Key Serial Number (for UKPT only)

Table 42 shows the format of a PIN profile.

Table 42. Format of a PIN Profile

Bytes	Description
0–7	PIN block format
8–15	Format control
16–23	Pad digit
24–47	Current Key Serial Number (for UKPT only)

PIN Block Format

This keyword specifies the format of the PIN block. The 8-byte value must be left-justified and padded with blanks. Refer to Table 43 for a list of valid values.

Table 43. Format Values of PIN Blocks

Format Value	Description
ECI-2	Eurocheque International format 2
ECI-3	Eurocheque International format 3
ISO-0	ISO format 0, ANSI X9.8, VISA 1, and ECI 1
ISO-1	ISO format 1 and ECI 4
ISO-2	ISO format 2
VISA-2	VISA format 2
VISA-3	VISA format 3
VISA-4	VISA format 4
3621	IBM 3621 and 5906
3624	IBM 3624
4704-EPP	IBM 4704 with encrypting PIN pad

Format Control

This keyword specifies whether there is any control on the user-supplied PIN format. The 8-byte value must be left-justified and padded with blanks. Specify one of the following values:

NONE No format control.

PBVC A PIN block variant constant (PBVC) enforces format control. Use the PBVC value only if you have coded PBVC in the encrypted PIN translate callable service. For the PBVC, the clear PIN key-encrypting key has been exclusive ORed with one of the PIN block formats. The cryptographic feature removes the pattern from the clear PIN key-encrypting key before it decrypts the PIN block.

Notes:

1. Only control vectors and extraction methods valid for the Cryptographic Coprocessor Feature may be used if the PBVC format control is desired.
2. PBVC is supported for compatibility with prior releases of OS/390 ICSF and existing ICSF applications. It is recommended that a format control of NONE be specified.

If you do not specify a value for the format control parameter, ICSF uses hexadecimal zeros.

Table 56 on page 225 lists the PIN block variant constants.

Pad Digit

Some PIN formats require this parameter. If the PIN format does not need a pad digit, the callable service ignores this parameter. Table 44 shows the format of a pad digit. The PIN profile pad digit must be specified in upper case.

Table 44. Format of a Pad Digit

Bytes	Description
16–22	Seven space characters
23	Character representation of a hexadecimal pad digit or a space if a pad digit is not needed. Characters must be one of the following: 0–9, A–F, or a blank.

Each PIN format supports only a pad digit in a certain range. The table below lists the valid pad digits for each PIN block format.

Table 45. Pad Digits for PIN Block Formats

PIN Block Format	Output PIN Profile	Input PIN Profile
ECI-2	Pad digit is not used	Pad digit is not used
ECI-3	Pad digit is not used	Pad digit is not used
ISO-0	F	Pad digit is not used
ISO-1	Pad digit is not used	Pad digit is not used
ISO-2	Pad digit is not used	Pad digit is not used
VISA-2	0 through 9	Pad digit is not used
VISA-3	0 through F	Pad digit is not used
VISA-4	F	Pad digit is not used
3621	0 through F	0 through F
3624	0 through F	0 through F
4704-EPP	F	Pad digit is not used

Recommendations for the Pad Digit

IBM recommends that you use a nondecimal pad digit in the range of A through F when processing IBM 3624 and IBM 3621 PIN blocks. If you use a decimal pad digit, the creator of the PIN block must ensure that the calculated PIN does not contain the pad digit, or unpredictable results may occur.

For example, you can exclude a specific decimal digit from being in any calculated PIN by using the IBM 3624 calculation procedure and by specifying a decimalization table that does not contain the desired decimal pad digit.

Current Key Serial Number

The current key serial number is the concatenation of the initial key serial number (a 59-bit value) and the encryption counter (a 21-bit value). The concatenation is an 80-bit (10-byte) value. Table 46 shows the format of the current key serial number.

Table 46. Format of a UKPT keyword

Bytes	Description
24–33	The current key serial number used to derive the initial PIN encrypting key.
34–47	Ignored.

Clear PIN Encrypt (CSNBCPE)

The Clear PIN Encrypt callable service formats a PIN into one of the following PIN block formats and encrypts the results. You can use this service to create an encrypted PIN block for transmission. With the RANDOM keyword, you can have the service generate random PIN numbers.

Note: A clear PIN is a sensitive piece of information. Ensure that your application program and system design provide adequate protection for any clear PIN value.

- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format
- IBM 4704 encrypting PINPAD (4704-EPP) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI2 format
- ECI3 format

ICSF routes the Clear PIN Encrypt request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBCPE(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             PIN_encrypting_key_identifier,
             rule_array_count,
             rule_array,
             clear_PIN,
             PIN_profile,
             PAN_data,
             sequence_number
             encrypted_PIN_block )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

PIN_encrypting_key_identifier

Direction: Input/Output Type: String

The 64-byte string containing an internal key token or a key label of an internal key token. The internal key token contains the key that encrypts the PIN block. The control vector in the internal key token must specify an OPINENC key type and have the CPINENC usage bit set to 1.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter.

rule_array

Direction: Input Type: Character string

Keywords that provide control information to the callable service. The keyword is left-justified in an 8-byte field, and padded on the right with blanks. All keywords must be in contiguous storage. The rule array keywords are shown as follows:

Table 47. Process Rules for the Clear PIN Encryption Callable Service

Process Rule	Description
ENCRYPT	This is the default. Use of this keyword is optional.
RANDOM	Causes the service to generate a random PIN value. The length of the PIN is based on the value in the *clear_PIN* variable. Set the value of the clear PIN to zero and use as many digits as the desired random PIN; pad the remainder of the clear PIN variable with space characters.

clear_PIN

Direction: Input	Type: String

A 16-character string with the clear PIN. The value in this variable must be left-justified and padded on the right with space characters.

PIN_profile

Direction: Input	Type: String

A 24-byte string containing three 8-byte elements with a PIN block format keyword, a format control keyword (NONE), and a pad digit as required by certain formats. See "The PIN Profile" on page 212 for additional information.

PAN_data

Direction: Input	Type: String

A 12-byte PAN in character format. The service uses this parameter if the PIN profile specifies the ISO-0 or VISA-4 keyword for the PIN block format. Otherwise, ensure that this parameter is a 12-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the ISO-0 keyword, use the 12 rightmost digits of the PAN data, excluding the check digit. When using the VISA-4 keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.

sequence_number

Direction: Input	Type: Integer

The 4-byte character integer. The service currently ignores the value in this variable. For future compatibility, the suggested value is 99999.

encrypted_PIN_block

Direction: Output	Type: String

The field that receives the 8-byte encrypted PIN block.

Restrictions

The caller must be in task mode, not in SRB mode.

The format control specified in the PIN profile must be NONE. If PBVC is specified as the format control, the service will fail.

Usage Note

SAF will be invoked to check authorization to use the Clear PIN encrypt service and the label of the *PIN_encrypting_key_identifier*.

Clear PIN Generate (CSNBPGN)

Use the Clear PIN generate callable service to generate a clear PIN, a PIN validation value (PVV), or an offset according to an algorithm. You supply the algorithm or process rule using the *rule_array* parameter.

- IBM 3624 (IBM-PIN or IBM-PINO)
- IBM German Bank Pool (GBP-PIN or GBP-PINO)
- VISA PIN validation value (VISA-PVV)
- Interbank PIN (INBK-PIN)

The callable service can execute only when ICSF is in special secure mode. This mode is described in “Special Secure Mode” on page 9.

For guidance information about VISA, see their appropriate publications. The Interbank PIN algorithm is available only on S/390 Enterprise Servers, the S/390 Multiprise, and the IBM @server Zseries.

ICSF routes the Clear PIN Generate request to a PCI Cryptographic Coprocessor if the control vector of the PIN generating key cannot be processed on the Cryptographic Coprocessor Feature. If no PCI Cryptographic Coprocessor is online in this case, the request fails.

Format

```
CALL CSNBPGN(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            PIN_generating_key_identifier,
            rule_array_count,
            rule_array,
            PIN_length,
            PIN_check_length,
            data_array,
            returned_result )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output | Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output | Type: String

The data that is passed to the installation exit.

PIN_generating_key_identifier

Direction: Input/Output | Type: Character string

The 64-byte key label or internal key token that identifies the PIN generation (PINGEN) key. If the *PIN_generating_key_identifier* identifies a key which does not have the default PIN generation key control vector, the request will be routed to a PCI Cryptographic Coprocessor.

rule_array_count

Direction: Input | Type: Integer

The number of process rules specified in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input | Type: Character string

The process rule provides control information to the callable service. Specify one of the values in Table 48. The keyword is left-justified in an 8-byte field, and padded on the right with blanks.

Table 48. Process Rules for the Clear PIN Generate Callable Service

Process Rule	Description
GBP-PIN	The IBM German Bank Pool PIN, which uses the institution PINGEN key to generate an institution PIN (IPIN).
GBP-PINO	The IBM German Bank Pool PIN offset, which uses the pool PINGEN key to generate a pool PIN (PPIN). It uses the institution PIN (IPIN) as input and calculates the PIN offset, which is the output.
IBM-PIN	The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.

Table 48. Process Rules for the Clear PIN Generate Callable Service (continued)

Process Rule	Description
IBM-PINO	The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset (the output).
INBK-PIN	The Interbank PIN is generated.
VISA-PVV	The VISA PIN validation value. Input is the customer PIN.

PIN_length

Direction: Input Type: Integer

The length of the PIN used for the IBM algorithms only, IBM-PIN or IBM-PINO. Otherwise, this parameter is ignored. Specify an integer from 4 through 16. If the length is greater than 12, the request will be routed to the PCI Cryptographic Coprocessor.

PIN_check_length

Direction: Input Type: Integer

The length of the PIN offset used for the IBM-PINO process rule only. Otherwise, this parameter is ignored. Specify an integer from 4 through 16.

Note: The PIN check length must be less than or equal to the integer specified in the *PIN_length* parameter.

data_array

Direction: Input Type: String

Three 16-byte data elements required by the corresponding *rule_array* parameter. The data array consists of three 16-byte fields or elements whose specification depends on the process rule. If a process rule only requires one or two 16-byte fields, then the rest of the data array is ignored by the callable service. Table 49 describes the array elements.

Table 49. Array Elements for the Clear PIN Generate Callable Service

Array Element	Description
Clear_PIN	Clear user selected PIN of 4 to 12 digits of 0 through 9. Left-justified and padded with spaces. For IBM-PINO, this is the clear customer PIN (CSPIN). For GBP-PINO, this is the institution PIN. For IBM-PIN and GBP-PIN, this field is ignored.
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen digits of 0 through 9.

Table 49. Array Elements for the Clear PIN Generate Callable Service (continued)

Array Element	Description
Trans_sec_parm	For VISA only, the leftmost sixteen digits. Eleven digits of the personal account number (PAN). One digit key index. Four digits of customer selected PIN. For Interbank only, sixteen digits. Eleven right-most digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.
Validation_data	Validation data for IBM and IBM German Bank Pool padded to 16 bytes. One to sixteen characters of hexadecimal account data left-justified and padded on the right with blanks.

Table 50 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 50. Array Elements Required by the Process Rule

Process Rule	IBM-PIN	IBM-PINO	GBP-PIN	GBP-PINO	VISA-PVV	INBK-PIN
Decimalization_table	1	1	1	1		
Validation_data	2	2	2	2		
Clear_PIN		3		3		
Trans_sec_parm					1	1

Note: Generate offset for GBP algorithm is equivalent to IBM offset generation with *PIN_check_length* of 4 and *PIN_length* of 6.

returned_result

Direction: Output Type: Character string

The 16-byte generated output, left-justified and padded on the right with blanks.

Restriction

PIN lengths of 13-16 require the optional PCI Cryptographic Coprocessor.

Usage Note

If you are using the IBM 3624 PIN and IBM German Bank Pool PIN algorithms, you can supply an unencrypted customer selected PIN to generate a PIN offset.

Related Information

PIN algorithms are shown in PIN Formats and Algorithms.

Clear PIN Generate Alternate (CSNBCPA)

Use the clear PIN generate alternate service to generate a clear VISA PVV (PIN validation value) from an input encrypted PIN block, or to produce a 3624 offset from a customer-selected encrypted PIN. The PIN block can be encrypted under either an input PIN-encrypting key (IPINENC) or an output PIN-encrypting key (OPINENC). Using an input PIN-encrypting key requires that the NOCV-enablement keys be present in the CKDS.

If the *PIN_encryption_key_identifier* identifies a key which does not have the default PIN encrypting control vector (either IPINENC or OPINENC), the request will be routed to a PCI Cryptographic Coprocessor for processing. If the IBM-PINO PIN calculation method is specified, the request will be routed to a PCI Cryptographic Coprocessor for processing. If PBVC is specified for format control, the request will be routed to the Cryptographic Coprocessor Feature. If anything is specified other than the default in the PIN extraction method keyword for the given PIN block format in *rule_array*, the request will be routed to the PCI Cryptographic Coprocessor.

Format

```
CALL CSNBCPA(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            PIN_encryption_key_identifier,
            PIN_generation_key_identifier,
            PIN_profile,
            PAN_data,
            encrypted_PIN_block,
            rule_array_count,
            rule_array,
            PIN_check_length,
            data_array,
            returned_PVV)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

PIN_encryption_key_identifier

Direction: Input/Output	Type: String

A 64-byte string consisting of an internal token that contains an IPINENC or OPINENC key or the label of an IPINENC or OPINENC key that is used to encrypt the PIN block. If you specify a label, it must resolve uniquely to either an IPINENC or OPINENC key. If the *PIN_encryption_key_identifier* identifies a key which does not have the default PIN encrypting control vector (either IPINENC or OPINENC), the request will be routed to the PCI Cryptographic Coprocessor for processing.

PIN_generation_key_identifier

Direction: Input/Output	Type: String

A 64-byte string that consists of an internal token that contains a PIN generation (PINGEN) key or the label of a PINGEN key. If the *PIN_generation_key_identifier* identifies a key which does not have the default PIN generating control vector, the request will be routed to the PCI Cryptographic Coprocessor for processing.

PIN_profile

Direction: Input	Type: Character string

The three 8-byte character elements that contain information necessary to extract a PIN from a formatted PIN block. The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the clear PIN generate alternate service. See “The PIN Profile” on page 212 for additional information.

PAN_data

Direction: Input	Type: String

A 12-byte field that contains 12 characters of PAN data. The personal account number recovers the PIN from the PIN block if the PIN profile specifies ISO-0 or VISA-4 block formats. Otherwise it is ignored, but you must specify this parameter.

For ISO-0, use the rightmost 12 digits of the PAN, excluding the check digit. For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

encrypted_PIN_block

Direction: Input	Type: String

An 8-byte field that contains the encrypted PIN that is input to the VISA PVV generation algorithm. The service uses the IPINENC or OPINENC key that is specified in the *PIN_encryption_key_identifier* parameter to encrypt the block.

rule_array_count

Direction: Input	Type: Integer

The number of process rules specified in the *rule_array* parameter. The value may be 1 or 2. If the default extraction method for a PIN block format is desired, you may code the rule array count value as 1.

rule_array

Direction: Input
Type: Character string

The process rule for the PIN generation algorithm. Specify IBM-PINO or “VISA-PVV” (the VISA PIN verification value) in an 8-byte field, left-justified, and padded with blanks. The *rule_array* points to an array of one or two 8-byte elements as follows:

Table 51. Rule Array Elements for the Clear PIN Generate Alternate Service

Rule Array Element	Function of Rule Array keyword
1	PIN calculation method
2	PIN extraction method

The first element in the rule array must specify one of the keywords that indicate the PIN calculation method as shown below:

Table 52. Rule Array Keywords (First Element) for the Clear PIN Generate Alternate Service

PIN Calculation Method Keyword	Meaning
IBM-PINO	This keyword specifies use of the IBM 3624 PIN Offset calculation method.
VISA-PVV	This keyword specifies use of the VISA PVV calculation method.

If the second element in the rule array is provided, it must specify one of the PIN extraction method keywords by PIN block format as shown below. The keyword listed first in a list is the default. If the default extraction method for a PIN block format is desired, you may code the rule array count value as 1.

Table 53. PIN Extraction Method Keywords

PIN Block Format	PIN Extraction Method Keywords
ECI-2	PINLEN04
ECI-3	PINBLOCK
ISO-0	PINBLOCK
ISO-1	PINBLOCK
ISO-2	PINBLOCK
VISA-2	PINBLOCK
VISA-3	PINBLOCK
VISA-4	PINBLOCK
3621	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN12, PADEXIST
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST

Table 53. PIN Extraction Method Keywords (continued)

PIN Block Format	PIN Extraction Method Keywords
4704-EPP	PINBLOCK

The PIN extraction methods operate as follows:

PINBLOCK
Specifies that the service use one of the following:
- the PIN length, if the PIN block contains a PIN length field
- the PIN delimiter character, if the PIN block contains a PIN delimiter character.

PADDIGIT
Specifies that the service use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT
Specifies that the service use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length.

PINLENxx
Specifies that the service use the length specified in the keyword, where xx can range from 4 to 16 digits, to identify the PIN.

PADEXIST
Specifies that the service use the character in the 16th position of the PIN block as the value of the pad value.

PIN_check_length

Direction: Input Type: Integer

The length of the PIN offset used for the IBM-PINO process rule only. Otherwise, this parameter is ignored. Specify an integer from 4 through 16.

Note: The PIN check length must be less than or equal to the integer specified in the *PIN_length* parameter.

data_array

Direction: Input Type: String

Three 16-byte elements. Table 54 describes the format when IBM-PINO is specified. Table 55 on page 225 describes the format when VISA-PVV is specified.

Table 54. Data Array Elements for the Clear PIN Generate Alternate Service (IBM-PINO)

Array Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 to 9) that are used to convert hexadecimal digits (X'0' to X'F') of the enciphered validation data to the decimal digits X'0' to X'9').
validation_data	This element contains one to 16 characters of account data. The data must be left justified and padded on the right with space characters.

Table 54. Data Array Elements for the Clear PIN Generate Alternate Service (IBM-PINO) (continued)

Array Element	Description
Reserved-3	This field is ignored, but you must specify it.

Table 55. Data Array Elements for the Clear PIN Generate Alternate Service (VISA-PVV)

Array Element	Description
Trans_sec_parm	For VISA-PVV only, the leftmost twelve digits. Eleven digits of the personal account number (PAN). One digit key index. The rest of the field is ignored.
Reserved-2	This field is ignored, but you must specify it.
Reserved-3	This field is ignored, but you must specify it.

returned_PVV

Direction: Output Type: Character

A 16-byte area that contains the 4-byte PVV left-justified and padded with blanks.

Restriction

The IBM-PINO PIN calculation method requires the optional PCI Cryptographic Coprocessor.

Usage Notes

To use an IPINENC key, you must install the NOCV-enablement keys in the CKDS.

The following table lists the PIN block variant constants (PBVC) to use.

Note: PBVC is supported for compatibility with prior releases of OS/390 ICSF and existing ICSF applications. If PBVC is specified in the format control parameter of the PIN profile, the Clear PIN Generate Alternate service will not be routed to a PCI Cryptographic Coprocessor for processing. This means that only control vectors and extraction methods valid for the Cryptographic Coprocessor Feature may be used if PBVC formatting is desired. It is recommended that a format control of NONE be used for maximum flexibility.

Table 56. PIN Block Variant Constants (PBVCs)

PIN Format Name	PIN Block Variant Constant (PBVC)
ECI-2	X'00000000000093000000000000009300'
ECI-3	X'00000000000095000000000000009500'
ISO-0	X'00000000000088000000000000008800'
ISO-1	X'0000000000008B000000000000008B00'
VISA-2	X'0000000000008D000000000000008D00'
VISA-3	X'0000000000008E000000000000008E00'
VISA-4	X'00000000000090000000000000009000'
3621	X'00000000000084000000000000008400'
3624	X'00000000000082000000000000008200'

Table 56. PIN Block Variant Constants (PBVCs) (continued)

PIN Format Name	PIN Block Variant Constant (PBVC)
4704-EPP	X'00000000000087000000000000008700'

Encrypted PIN Generate (CSNBEPG)

The Encrypted PIN Generate callable service formats a PIN and encrypts the PIN block. To generate the PIN, the service uses one of the following PIN calculation methods:

- IBM 3624 PIN
- IBM German Bank Pool Institution PIN
- Interbank PIN

To format the PIN, the service uses one of the following PIN block formats:

- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI-1 formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format
- IBM 4704 encrypting PINPAD (4704-EPP) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI-2 format
- ECI-3 format

ICSF routes the Encrypted PIN Generate request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNBEPG(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            PIN_generating_key_identifier,
            outbound_PIN_encrypting_key_identifier
            rule_array_count,
            rule_array,
            PIN_length,
            data_array,
            PIN_profile,
            PAN_data,
            sequence_number
            encrypted_PIN_block )
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFFF' (2 gigabytes). The data is defined in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

PIN_generating_key_identifier

Direction: Input/Output Type: String

The 64-byte internal key token or a key label of an internal key token in the CKDS. The internal key token contains the PIN-generating key. The control vector must specify the PINGEN key type and have the EPINGEN usage bit set to 1.

outbound_PIN_encrypting_key_identifier

Direction: Input Type: String

A 64-byte internal key token or a key label of an internal key token in the CKDS. The internal key token contains the key to be used to encrypt the formatted PIN and must contain a control vector that specifies the OPINENC key type and has the EPINGEN usage bit set to 1.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input Type: Character string

Keywords that provide control information to the callable service. Each keyword is left-justified in an 8-byte field, and padded on the right with blanks. All keywords must be in contiguous storage. The rule array keywords are shown as follows:

Table 57. Process Rules for the Encrypted PIN Generate Callable Service

Process Rule	Description
GBP-PIN	This keyword specifies the IBM German Bank Pool Institution PIN calculation method is to be used to generate a PIN.
IBM-PIN	This keyword specifies the IBM 3624 PIN calculation method is to be used to generate a PIN.
INBK-PIN	This keyword specifies the Interbank PIN calculation method is to be used to generate a PIN.

PIN_length

Direction: Input Type: Integer

A integer defining the PIN length for those PIN calculation methods with variable length PINs; otherwise, the variable should be set to zero.

data_array

Direction: Input Type: String

Three 16-byte character strings, which are equivalent to a single 48-byte string. The values in the data array depend on the keyword for the PIN calculation method. Each element is not always used, but you must always declare a complete data array. The numeric characters in each 16-byte string must be from 1 to 16 bytes in length, uppercase, left-justified, and padded on the right with space characters. Table 58 describes the array elements.

Table 58. Array Elements for the Encrypted PIN Generate Callable Service

Array Element	Description
Clear_PIN	Clear user selected PIN of 4 to 12 digits of 0 through 9. Left-justified and padded with spaces. For IBM-PINO, this is the clear customer PIN (CSPIN). For GBP-PINO, this is the institution PIN. For IBM-PIN and GBP-PIN, this field is ignored.
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen characters that are used to map the hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9').
Trans_sec_parm	For Interbank only, sixteen digits. Eleven right-most digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.
Validation_data	Validation data for IBM and IBM German Bank Pool padded to 16 bytes. One to sixteen characters of hexadecimal account data left-justified and padded on the right with blanks.

Table 59 on page 229 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 59. Array Elements Required by the Process Rule

Process Rule	IBM-PIN	IBM-PINO	GBP-PIN	GBP-PINO	VISA-PVV	INBK-PIN
Decimalization_table	1	1	1	1		
Validation_data	2	2	2	2		
Clear_PIN		3		3		
Trans_sec_parm					1	1

PIN_profile

Direction: Input Type: String array

A 24-byte string containing the PIN profile including the PIN block format. See "The PIN Profile" on page 212 for additional information.

PAN_data

Direction: Input Type: String

A 12-byte string that contains 12 digits of Personal Account Number (PAN) data. The service uses this parameter if the PIN profile specifies the ISO-0 or VISA-4 keyword for the PIN block format. Otherwise, ensure that this parameter is a 4-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the ISO-0 keyword, use the 12 rightmost digit of the PAN data, excluding the check digit. When using the VISA-4 keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.

sequence_number

Direction: Input Type: Integer

The 4-byte string that contains the sequence number used by certain PIN block formats. The service uses this parameter if the PIN profile specifies the 3621 or 4704-EPP keyword for the PIN block format. Otherwise, ensure that this parameter is a 4-byte variable in application data storage. The information in the variable will be ignored, but the variable must be declared. To enter a sequence number, do the following:

- Enter 99999 to use a random sequence number that the service generates.
- For the 3621 PIN block format, enter a value in the range from 0 to 65535.
- For the 4704-EPP PIN block format, enter a value in the range from 0 to 255.

encrypted_PIN_block

Direction: Output Type: String

The field where the service returns the 8-byte encrypted PIN.

Restrictions

The caller must be in task mode, not in SRB mode.

The format control specified in the PIN profile must be NONE. If PBVC is specified as the format control, the service will fail.

Usage Note

SAF will be invoked to check authorization to use the Encrypted PIN Generate service and any key labels specified as input.

Encrypted PIN Translate (CSNBPTR)

Use the encrypted PIN translate callable service to reencipher a PIN block from one PIN-encrypting key to another and, optionally, to change the PIN block format, such as the pad digit or sequence number.

The encrypted PIN translate service can be used for unique-key-per-transaction (UKPT) key derivation. UKPT is supported for the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* parameters for both REFORMAT and TRANSLAT process rules. The *rule_array* keyword determines which PIN key(s) are derived UKPT key(s).

ICSF routes the encrypted PIN translate request to a PCI Cryptographic Coprocessor if the control vector in a supplied PIN encrypting key cannot be processed on the Cryptographic Coprocessor Feature. UKPT support requires a PCI Cryptographic Coprocessor. If one is not active, the request fails. The request is also routed to a PCI Cryptographic Coprocessor if the PIN profile specifies the ISO-2 PIN block format. If no PCI Cryptographic Coprocessor is active in this case, the request fails. If PBVC is specified for format control, the request will be routed to the Cryptographic Coprocessor Feature. If anything is specified other than the default in the PIN extraction method keyword for the given PIN block format in *rule_array*, the request will be routed to the PCI Cryptographic Coprocessor.

Format

```
CALL CSNBPTR(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            input_PIN_encrypting_key_identifier,
            output_PIN_encrypting_key_identifier,
            input_PIN_profile,
            PAN_data_in,
            PIN_block_in,
            rule_array_count,
            rule_array,
            output_PIN_profile,
            PAN_data_out,
            sequence_number,
            PIN_block_out )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output　　Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output　　Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output　　Type: String

The data that is passed to the installation exit.

input_PIN_encrypting_key_identifier

Direction: Input/Output　　Type: String

The input PIN-encrypting key (IPINENC) for the *PIN_block_in* parameter specified as a 64-byte internal key token or a key label. If keyword UKPTIPIN or UKPTBOTH is specified in the *rule_array*, then the *input_PIN_encrypting_key_identifier* must specify a key token or key label of a KEYGENKY with the UKPT usage bit enabled. If the *input_PIN_encrypting_key_identifier* identifies a key which does not have the default input PIN encrypting key control vector (IPINENC), the request will be routed to a PCI Cryptographic Coprocessor.

output_PIN_encrypting_key_identifier

Direction: Input/Output　　Type: String

The output PIN-encrypting key (OPINENC) for the *PIN_block_out* parameter specified as a 64-byte internal key token or a key label. If keyword UKPTOPIN or UKPTBOTH is specified in the *rule_array*, then the *output_PIN_encrypting_key_identifier* must specify a key token or key label of a KEYGENKY with the UKPT usage bit enabled. If the *output_PIN_encrypting_key_identifier* identifies a key which does not have the default output PIN encrypting key control vector (OPINENC), the request will be routed to a PCI Cryptographic Coprocessor.

input_PIN_profile

Direction: Input　　Type: Character string

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the callable service. See “The PIN Profile” on page 212 for additional information.

If you choose the TRANSLAT processing rule in the *rule_array* parameter, the *input_PIN_profile* and the *output_PIN_profile* must specify the same PIN block format. If you choose the REFORMAT processing rule in the *rule_array* parameter, the input PIN profile and output PIN profile can have different PIN block formats. If you specify UKPTIPIN or UKPTBOTH in the *rule_array* parameter, then the *input_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See “The PIN Profile” on page 212 for additional information.

The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Encrypted PIN translate callable service with a process rule (*rule_array* parameter) of REFORMAT. If the process rule is TRANSLAT, the pad digit is ignored.

PAN_data_in

Direction: Input   Type: Character string

The personal account number (PAN) if the process rule (*rule_array* parameter) is REFORMAT and the input PIN format is ISO-0 or VISA-4 only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For ISO-0, use the rightmost 12 digits of the PAN, excluding the check digit.

For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

PIN_block_in

Direction: Input   Type: String

The 8-byte enciphered PIN block that contains the PIN to be translated.

rule_array_count

Direction: Input   Type: Integer

The number of process rules specified in the *rule_array* parameter. The value may be 1, 2 or 3.

rule_array

Direction: Input   Type: Character string

The process rule for the callable service.

Table 60. Keywords for Encrypted PIN Translate

Keyword	Meaning
Processing Rules (required)	
REFORMAT	Changes the PIN format, the contents of the PIN block, and the PIN-encrypting key.

Table 60. Keywords for Encrypted PIN Translate (continued)

Keyword	Meaning	
TRANSLAT	Changes the PIN-encrypting key only. It does not change the PIN format and the contents of the PIN block.	
PIN Block Format and PIN Extraction Method (optional) **Note:** If a PIN extraction method is not specified, the first one listed for each PIN block format will be the default.		
ECI-2	PINLEN04	The PIN extraction method keywords specify a PIN extraction method for a PINLEN04 format.
ECI-3	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-0	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-1	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-2	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-2	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-3	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-4	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
3621	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN12, PADEXIST	The PIN extraction method keywords specify a PIN extraction method for an IBM 3621 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST	The PIN extraction method keywords specify a PIN extraction method for an IBM 3624 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.

Table 60. Keywords for Encrypted PIN Translate (continued)

Keyword	Meaning	
4704-EPP	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
UKPT Keywords (optional)		
UKPTIPIN	The *input_PIN_encrypting_key_identifier* is to be derived using the UKPT algorithm. The *input_PIN_encrypting_key_identifier* must be a KEYGENKY key with the UKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.	
UKPTOPIN	The *output_PIN_encrypting_key_identifier* is to be derived using the UKPT algorithm. The *output_PIN_encrypting_key_identifier* must be a KEYGENKY key with the UKPT usage bit enabled. The *output_PIN_profile* must be 48 bytes and contain the key serial number.	
UKPTBOTH	Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* are derived using the UKPT algorithm. Both the *input_PIN_encrypting_key_identifier* and the *output_PIN_encrypting_key_identifier* must be KEYGENKY keys with the UKPT usage bit enabled. Both the *input_PIN_profile* and the *output_PIN_profile* must be 48 bytes and contain the respective key serial number.	

output_PIN_profile

Direction: Input Type: Character string

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile, depending on whether the PIN block is being enciphered or deciphered by the callable service.

If you choose the TRANSLAT processing rule in the *rule_array* parameter, the *input_PIN_profile* and the *output_PIN_profile* must specify the same PIN block format. If you choose the REFORMAT processing rule in the *rule_array* parameter, the input PIN profile and output PIN profile can have different PIN block formats. If you specify UKPTOPIN or UKPTBOTH in the *rule_array* parameter, then the *output_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See "The PIN Profile" on page 212 for additional information.

PAN_data_out

Direction: Input Type: Character string

The personal account number (PAN) if the process rule (*rule_array* parameter) is REFORMAT and the output PIN format is ISO-0 or VISA-4 only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For ISO-0, use the rightmost 12 digits of the PAN, excluding the check digit.

For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

sequence_number

Direction: Input	Type: Integer

The sequence number if the process rule (*rule_array* parameter) is REFORMAT and the output PIN block format is 3621 or 4704-EPP only. Specify the integer value 99999. Otherwise, this parameter is ignored.

PIN_block_out

Direction: Output	Type: String

The 8-byte output PIN block that is reenciphered.

Restriction

Use of the ISO-2 PIN block format requires the optional PCI Cryptographic Coprocessor.

Use of the UKPT keywords require the optional PCI Cryptographic Coprocessor.

Usage Notes

Some PIN block formats are known by several names. The following table shows the additional names.

Table 61. Additional Names for PIN Formats

PIN Format	Additional Name
ISO-0	ANSI X9.8, VISA format 1, ECI format 1
ISO-1	ECI format 4

The following table lists the PIN block variant constants (PBVC) to be used.

Note: PBVC is supported for compatibility with prior releases of OS/390 ICSF and existing ICSF applications. If PBVC is specified in the format control parameter of the PIN profile, the Encrypted PIN translate service will not be routed to a PCI Cryptographic Coprocessor for processing. This means that only control vectors and extraction methods valid for the Cryptographic Coprocessor Feature may be used if PBVC formatting is desired. It is recommended that a format control of NONE be used for maximum flexibility.

Table 62. PIN Block Variant Constants (PBVCs)

PIN Format Name	PIN Block Variant Constant (PBVC)
ECI-2	X'00000000000093000000000000009300'
ECI-3	X'00000000000095000000000000009500'
ISO-0	X'00000000000088000000000000008800'
ISO-1	X'0000000000008B000000000000008B00'
VISA-2	X'0000000000008D000000000000008D00'
VISA-3	X'0000000000008E000000000000008E00'
VISA-4	X'00000000000090000000000000009000'

Table 62. PIN Block Variant Constants (PBVCs) (continued)

PIN Format Name	PIN Block Variant Constant (PBVC)
3621	X'00000000000084000000000000008400'
3624	X'00000000000082000000000000008200'
4704-EPP	X'00000000000087000000000000008700'

Encrypted PIN Verify (CSNBPVR)

Use the Encrypted PIN verify callable service to verify that one of the following customer selected trial PINs is valid:

- IBM 3624 (IBM-PIN)
- IBM 3624 PIN offset (IBM-PINO)
- IBM German Bank Pool (GBP-PIN)
- IBM German Bank Pool PIN offset (GBP-PINO)
- VISA PIN validation value (VISA-PVV)
- Interbank PIN (INBK-PIN)

| UKPT is supported for the Encrypted PIN Verify service through the *input_PIN_encrypting_key_identifier* parameter and keyword UKPTIPIN.

ICSF routes the Encrypted PIN verify request to a PCI Cryptographic Coprocessor if the control vector in a supplied PIN key cannot be processed on the Cryptographic Coprocessor Feature. The request is also routed to a PCI Cryptographic Coprocessor if the PIN profile specifies the ISO-2 PIN block format. If no PCI Cryptographic Coprocessor is online in this case, the request fails. If PBVC is specified for format control, the request will be routed to the Cryptographic Coprocessor Feature. If anything is specified other than the default in the PIN extraction method keyword for the given PIN block format in *rule_array*, the request will be routed to the PCI Cryptographic Coprocessor.

| UKPT support requires a PCI Cryptographic Coprocessor. If one is not active, the request fails.

Format

```
CALL CSNBPVR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             input_PIN_encrypting_key_identifier,
             PIN_verifying_key_identifier,
             input_PIN_profile,
             PAN_data,
             encrypted_PIN_block,
             rule_array_count,
             rule_array,
             PIN_check_length,
             data_array )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

input_PIN_encrypting_key_identifier

Direction: Input/Output	Type: String

The 64-byte key label or internal key token containing the PIN-encrypting key (IPINENC) that enciphers the PIN block. If the *input_PIN_encrypting_key_identifier* identifies a key which does not have the default PIN encrypting key control vector (IPINENC), the request will be routed to a PCI Cryptographic Coprocessor. If keyword UKPTIPIN is specified in the *rule_array*, then the *input_PIN_encrypting_key_identifier* must specify a key token or key label of a KEYGENKY with the UKPT usage bit enabled.

PIN_verifying_key_identifier

Direction: Input/Output	Type: String

The 64-byte key label or internal key token that identifies the PIN verify (PINVER) key. If the *PIN_verifying_key_identifier* identifies a key which does not have the default PIN verify key control vector, the request will be routed to a PCI Cryptographic Coprocessor.

input_PIN_profile

Direction: Input	Type: Character string

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the callable service. If you specify UKPTIPIN in the *rule_array* parameter, then the *input_PIN_profile* is extended to a 48-byte field and must contain the current key serial number. See “The PIN Profile” on page 212 for additional information.

The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the encrypted PIN verify callable service.

PAN_data

Direction: Input Type: Character string

The personal account number (PAN) is required for ISO-0 and VISA-4 only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For ISO-0, use the rightmost 12 digits of the PAN, excluding the check digit.

For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

encrypted_PIN_block

Direction: Input Type: String

The 8-byte enciphered PIN block that contains the PIN to be verified.

rule_array_count

Direction: Input Type: Integer

The number of process rules specified in the *rule_array* parameter. The value may be 1, 2 or 3.

rule_array

Direction: Input Type: Character string

The process rule for the PIN verify algorithm.

Table 63. Keywords for Encrypted PIN Verify

Keyword	Meaning
Algorithm Value (required)	
GBP-PIN	The IBM German Bank Pool PIN. It verifies the PIN entered by the customer and compares that PIN with the institution generated PIN by using an institution key.
GBP-PINO	The IBM German Bank Pool PIN offset. It verifies the PIN entered by the customer by comparing with the calculated institution PIN (IPIN) and adding the specified offset to the pool PIN (PPIN) generated by using a pool key.
IBM-PIN	The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.
IBM-PINO	The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset.
INBK-PIN	The Interbank PIN verify algorithm.

Table 63. Keywords for Encrypted PIN Verify (continued)

Keyword	Meaning	
VISA-PVV	The VISA PIN verify value.	
PIN Block Format and PIN Extraction Method (optional) **Note:** If a PIN extraction method is not specified, the first one listed for each PIN block format will be the default.		
ECI-2	PINLEN04	The PIN extraction method keywords specify a PIN extraction method for a PINLEN04 format.
ECI-3	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-0	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-1	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
ISO-2	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-2	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-3	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
VISA-4	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
3621	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN12, PADEXIST	The PIN extraction method keywords specify a PIN extraction method for an IBM 3621 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST	The PIN extraction method keywords specify a PIN extraction method for an IBM 3624 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.

Table 63. Keywords for Encrypted PIN Verify (continued)

Keyword	**Meaning**	
4704-EPP	PINBLOCK	The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.
UKPT Keywords (optional)		
UKPTIPIN	The *input_PIN_encrypting_key_identifier* is to be derived using the UKPT algorithm. The *input_PIN_encrypting_key_identifier* must be a KEYGENKY key with the UKPT usage bit enabled. The *input_PIN_profile* must be 48 bytes and contain the key serial number.	

PIN_check_length

Direction: Input   Type: Integer

The PIN check length for the IBM-PIN or IBM-PINO process rules only. Otherwise, it is ignored. Specify the rightmost digits, 4 through 16, for the PIN to be verified.

data_array

Direction: Input   Type: String

Three 16-byte elements required by the corresponding *rule_array* parameter. The data array consists of three 16-byte fields whose specification depend on the process rule. If a process rule only requires one or two 16-byte fields, then the rest of the data array is ignored by the callable service. Table 64 describes the array elements.

Table 64. Array Elements for the Encrypted PIN Verify Callable Service

Array Element	**Description**
Decimalization_table	Decimalization table for IBM and GBP only. Sixteen decimal digits of 0 through 9.
PIN_offset	Offset data for IBM-PINO and GBP-PINO. One to twelve numeric characters, 0 through 9, left-justified and padded on the right with blanks. For IBM-PINO, the PIN offset length is specified in the *PIN_check_length* parameter. For GBP-PINO, the PIN offset is always 4 digits. For IBM-PIN and GBP-PIN, the field is ignored.
trans_sec_parm	For VISA, only the leftmost twelve digits of the 16-byte field are used. These consist of the rightmost eleven digits of the personal account number (PAN) and a one-digit key index. The remaining four characters are ignored. For Interbank only, all 16 bytes are used. These consist of the rightmost eleven digits of the PAN, a constant of X'6', a one-digit key index, and three numeric digits of PIN validation data.
RPVV	For VISA-PVV only, referenced PVV (4 bytes) that is left-justified. The rest of the field is ignored.

Table 64. Array Elements for the Encrypted PIN Verify Callable Service (continued)

Array Element	Description
Validation_data	Validation data for IBM and GBP padded to 16 bytes. One to sixteen characters of hexadecimal account data left-justified and padded on the right with blanks.

Table 65 lists the data array elements required by the process rule (*rule_array* parameter). The numbers refer to the process rule's position within the array.

Table 65. Array Elements Required by the Process Rule

Process Rule	IBM-PIN	IBM-PINO	GBP-PIN	GBP-PINO	VISA-PVV	INBK-PIN
Decimalization_table	1	1	1	1		
PIN_offset	3	3	3	3		
RPVV					2	
Trans_sec_parm					1	1
Validation_data	2	2	2	2		

Restrictions

GBP-PINO is only supported if the CSNBPVR service is processed on the Cryptographic Coprocessor Feature. If the service is routed to a PCI Cryptographic Coprocessor, the service request will fail if the GBP-PINO calculation method is specified.

Use of the ISO-2 PIN block format requires the optional PCI Cryptographic Coprocessor.

| Use of the UKPTIPIN keyword requires the optional PCI Cryptographic
| Coprocessor.

Usage Notes

The following table lists the PIN block variant constants (PBVC) to be used.

Note: PBVC is supported for compatibility with prior releases of OS/390 ICSF and existing ICSF applications. If PBVC is specified in the format control parameter of the PIN profile, the Encrypted PIN Verify service will not be routed to a PCI Cryptographic Coprocessor for processing. This means that only control vectors and extraction methods valid for the Cryptographic Coprocessor Feature may be used if PBVC formatting is desired. It is recommended that a format control of NONE be used for maximum flexibility.

Table 66. PIN Block Variant Constants (PBVCs)

PIN Format Name	PIN Block Variant Constant (PBVC)
ECI-2	X'00000000000093000000000000009300'
ECI-3	X'00000000000095000000000000009500'
ISO-0	X'00000000000088000000000000008800'
ISO-1	X'0000000000008B000000000000008B00'
VISA-2	X'0000000000008D000000000000008D00'
VISA-3	X'0000000000008E000000000000008E00'
VISA-4	X'00000000000090000000000000009000'
3621	X'00000000000084000000000000008400'

Table 66. PIN Block Variant Constants (PBVCs) (continued)

PIN Format Name	PIN Block Variant Constant (PBVC)
3624	X'00000000000082000000000000008200'
4704-EPP	X'00000000000087000000000000008700'

Related Information

PIN Formats and Algorithms discusses the PIN algorithms in detail.

Secure Messaging for Keys (CSNBSKY)

The Secure Messaging for Keys callable service will encrypt a text block including a clear key value decrypted from an internal or external DES token. The text block is normally a ″Value″ field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Processing for this service is routed to the PCI Cryptographic Coprocessor. Keys only appear in the clear within the secure boundary of the PCI Cryptographic Coprocessor, and never in host storage.

Format

```
CALL CSNBSKY(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            input_key_identifier,
            key_encrypting_key_identifier,
            secmsg_key_identifier,
            text_length,
            clear_text,
            initialization_vector,
            key_offset,
            key_offset_field_length,
            enciphered_text,
            output_chaining_vector )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The valid values are 0 and 1.

rule_array

Direction: Input Type: Character String

Keywords that provides control information to the callable service. The processing method is the encryption mode used to encrypt the message.

Table 67. Rule Array Keywords for Secure Messaging for Keys

Keyword	Meaning
Enciphering mode (optional)	
TDES-CBC	Use CBC mode to encipher the message (default).
TDES-ECB	Use EBC mode to encipher the message.

input_key_identifier

Direction: Input/Output Type: String

The internal token, external token, or key label of an internal token of a double length DES key. The key is recovered in the clear and placed in the text to be encrypted. The control vector of the DES key must not prohibit export.

key_encrypting_key_identifier

Direction: Input/Output Type: String

If the *input_key_identifier* is an external token, then this parameter is the internal token or the key label of the internal token of IMPORTER or EXPORTER. If it is not, it is a null token. If a key label is specified, the key label must be unique.

secmsg_key_identifier

Direction: Input/Output Type: String

The internal token or key label of a secure message key for encrypting keys. This key is used to encrypt the updated *clear_text* containing the recovered DES key.

text_length

Direction: Input   Type: Integer

The length of the *clear_text* parameter that follows. Length must be a multiple of eight. Maximum length is 4K.

clear_text

Direction: Input   Type: String

Clear text that contains the recovered DES key at the offset specified and is then encrypted. Any padding or formatting of the message must be done by the caller on input.

initialization_vector

Direction: Input   Type: String

The 8-byte supplied string for the TDES-CBC mode of encryption. The *initialization_vector* is XORed with the first 8 bytes of *clear_text* before encryption. This field is ignored for TDES-ECB mode.

key_offset

Direction: Input   Type: Integer

The offset within the *clear_text* parameter at *key_offset* where the recovered clear *input_key_identifier* value is to be placed. The first byte of the *clear_text* field is offset 0.

key_offset_field_length

Direction: Input   Type: Integer

The length of the field within *clear_text* parameter at *key_offset* where the recovered clear *input_key_identifier* value is to be placed. Length must be a multiple of eight and is equal to the key length of the recovered key. The key must fit entirely within the *clear_text*.

enciphered_text

Direction: Output   Type: String

The field where the enciphered text is returned. The length of this field must be at least as long as the *clear_text* field.

output_chaining_vector

Direction: Output   Type: String

This field contains the last 8 bytes of enciphered text and is used as the *initialization_vector* for the next encryption call if data needs to be chained for TDES-CBC mode. No data is returned for TDES-ECB.

Restrictions

- This service requires at least one PCI Cryptographic Coprocessor to be installed and active.
- Caller must be task mode and must not be SRB mode.

Usage Notes

- SAF will be invoked to check authorization to use the secure messaging for keys service and any key labels specified as input.

Secure Messaging for PINs (CSNBSPN)

The Secure Messaging for PINs callable service will encrypt a text block including a clear PIN block recovered from an encrypted PIN block. The input PIN block will be reformatted if the block format in the *input_PIN_profile* is different than the block format n the *output_PIN_profile*. The clear PIN block will only be self encrypted if the SELFENC keyword is specified in the *rule_array*. The text block is normally a ″Value″ field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Processing for this service is routed to the PCI Cryptographic Coprocessor. PINs only appear in the clear within the secure boundary of the PCI Cryptographic Coprocessor, and never in host storage.

Format

```
CALL CSNBSPN(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            input_PIN_block,
            PIN_encrypting_key_identifier,
            input_PIN_profile,
            input_PAN_data,
            secmsg_key_identifier,
            output_PIN_profile,
            output_PAN_data,
            text_length,
            clear_text,
            initialization_vector,
            PIN_offset,
            PIN_offset_field_length,
            enciphered_text,
            output_chaining_vector )
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output　　Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output　　Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output　　Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input　　Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The valid values are 0, 1, or 2.

rule_array

Direction: Input　　Type: Character String

Keywords that provide control information to the callable service. The processing method is the algorithm used to create the generated key. The keywords are left justified and padded on the right with blanks.

Table 68. Rule Array Keywords for Secure Messaging for PINs

Keyword	Meaning
Enciphering mode (optional)	
TDES-CBC	Use CBC mode to encipher the message (default).
TDES-ECB	Use EBC mode to encipher the message.
PIN encryption (optional)	
CLEARPIN	Recovered clear input PIN block (may be reformatted) is placed in the clear in the message for encryption with the secure message key (default).
SELFENC	Recovered clear input PIN block (may be reformatted) is self-encrypted and then placed in the message for encryption with the secure message key.

input_PIN_block

Direction: Input	Type: String

The 8-byte input PIN block that is to be recovered in the clear and perhaps reformatted, and then placed in the *clear_text* to be encrypted.

PIN_encrypting_key_identifier

Direction: Input/Output	Type: String

The internal token or key label of the internal token of the PIN encrypting key used in encrypting the *input_PIN_block*. The key must be an IPINENC key.

input_PIN_profile

Direction: Input	Type: Character String

The three 8-byte character elements that contain information necessary to extract the PIN from a formatted PIN block. The valid input PIN formats are ISO-0, ISO-1, and ISO-2. See "The PIN Profile" on page 212 for additional information.

input_PAN_data

Direction: Input	Type: Character String

The 12 digit personal account number (PAN) if the input PIN format is ISO-0 only. Otherwise, the parameter is ignored.

secmsg_key_identifier

Direction: Input/Output	Type: String

The internal token or key label of an internal token of a secure message key for encrypting PINs. This key is used to encrypt the updated *clear_text*.

output_PIN_profile

Direction: Input	Type: String

The three 8-byte character elements that contain information necessary to create a formatted PIN block. If reformatting is not required, the *input_PIN_profile* and the *output_PIN_profile* must specify the same PIN block format. Output PIN block formats supported are ISO-0, ISO-1, and ISO-2.

output_PAN_data

Direction: Input	Type: String

The 12 digit personal account number (PAN) if the output PIN format is ISO-0 only. Otherwise, this parameter is ignored.

text_length

Direction: Input	Type: Integer

The length of the *clear_text* parameter that follows. Length must be a multiple of eight. Maximum length is 4K.

clear_text

Direction: Input	Type: String

Clear text that contains the recovered and/or reformatted/encrypted PIN at offset specified and then encrypted. Any padding or formatting of the message must be done by the caller on input.

initialization_vector

Direction: Input	Type: String

The 8-byte supplied string for the TDES-CBC mode of encryption. The *initialization_vector* is XORed with the first 8 bytes of *clear_text* before encryption. This field is ignored for TDES-ECB mode.

PIN_offset

Direction: Input	Type: Integer

The offset within the *clear_text* parameter where the reformatted PIN block is to be placed. The first byte of the *clear_text* field is offset 0.

PIN_offset_field_length

Direction: Input	Type: Integer

The length of the field within *clear_text* parameter at *PIN_offset* where the recovered clear *input_PIN_block* value is to be placed. The PIN block may be self-encrypted if requested by the rule array. Length must be eight. The PIN block must fit entirely within the *clear_text*.

enciphered_text

Direction: Output	Type: String

The field where the enciphered text is returned. The length of this field must be at least as long as the *clear_text* field.

output_chaining_vector

Direction: Output	Type: String

This field contains the last 8 bytes of enciphered text and is used as the *initialization_vector* for the next encryption call if data needs to be chained for TDES-CBC mode. No data is returned for TDES-ECB.

Restrictions

- This service requires at least one PCI Cryptographic Coprocessor to be installed and active.
- Caller must be task mode and must not be SRB mode.

Usage Notes

- SAF will be invoked to check authorization to use the secure messaging for PINs service and any key labels specified as input.

SET Block Compose (CSNDSBC)

The SET Block Compose callable service performs DES-encryption of data, OAEP-formatting through a series of SHA-1 hashing operations, and the RSA-encryption of the Optimal Asymmetric Encryption Padding (OAEP) block.

This service routes the request to a PCI Cryptographic Coprocessor to perform the RSA-OAEP processing. If there are no PCI Cryptographic Coprocessors online, the request is routed to the Cryptographic Coprocessor Feature.

Format

```
CALL CSNDSBC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             block_contents_identifier,
             XData_string_length,
             XData_string,
             data_to_encrypt_length,
             data_to_encrypt,
             data_to_hash_length,
             data_to_hash,
             initialization_vector,
             RSA_public_key_identifier_length,
             RSA_public_key_identifier,
             DES_key_block_length,
             DES_key_block,
             RSA_OAEP_block_length,
             RSA_OAEP_block,
             chaining_vector,
             DES_encrypted_data_block )
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be at least 1.

rule_array

Direction: Input Type: Character String

Keywords that provides control information to the callable service. The keyword must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks.

Table 69. Keywords for SET Block Compose Control Information

Keyword	Meaning
Block Type (required)	
SET1.00	The structure of the RSA-OAEP encrypted block is defined by SET protocol.
Formatting Information (optional)	
DES-ONLY	DES encryption only is to be performed; no RSA-OAEP formatting will be performed. (See Usage Notes.)

block_contents_identifier

Direction: Input Type: String

A one-byte string, containing a binary value that will be copied into the Block Contents (BC) field of the SET DB data block (indicates what data is carried in the Actual Data Block, ADB, and the format of any extra data (*XData_string*)). This parameter is ignored if DES-ONLY is specified in the rule-array.

XData_string_length

Direction: Input	Type: Integer

The length in bytes of the data contained within *XData_string*. The maximum length is 94 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

XData_string

Direction: Input	Type: String

Extra-encrypted data contained within the OAEP-processed and RSA-encrypted block. The format is indicated by *block_contents_identifier*. For a *XData_string_length* value of zero, *XData_string* must still be specified, but will be ignored by ICSF. The string is treated as a string of hexadecimal digits. This parameter is ignored if DES-ONLY is specified in the rule-array.

data_to_encrypt_length

Direction: Input/Output	Type: Integer

The length in bytes of data that is to be DES-encrypted. The length has a maximum value of 32 MB minus 8 bytes to allow for up to 8 bytes of padding. The data is identified in the *data_to_encrypt* parameter. On output, this value is updated with the length of the encrypted data in the *DES_encrypted_data_block*.

data_to_encrypt

Direction: Input	Type: String

The data that is to be DES-encrypted (with a 64-bit DES key generated by this service). The data will be padded by this service according to the PKSC #5 padding rules.

data_to_hash_length

Direction: Input	Type: Integer

The length in bytes of the data to be hashed. The hash is an optional part of the OAEP block. If the *data_to_hash_length* is 0, no hash will be included in the OAEP block. This parameter is ignored if DES-ONLY is specified in the *rule_array* parameter.

data_to_hash

Direction: Input	Type: String

The data that is to be hashed and included in the OAEP block. No hash is computed or inserted in the OAEP block if the *data_to_hash_length* is 0. This parameter is ignored if DES-ONLY is specified in the *rule_array* parameter.

initialization_vector

Direction: Input Type: String

An 8-byte string containing the initialization vector to be used for the cipher block chaining for the DES encryption of the data in the *data_to_encrypt* parameter. The same initialization vector must be used to perform the DES decryption of the data.

RSA_public_key_identifier_length

Direction: Input Type: Integer

The length of the *RSA_public_key_identifier* field. The maximum size is 2500 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

RSA_public_key_identifier

Direction: Input Type: String

A string containing either the key label of the RSA public key or the RSA public key token to be used to perform the RSA encryption of the OAEP block. The modulus bit length of the key must be 1024 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

DES_key_block_length

Direction: Input/Output Type: Integer

The length of the *DES_key_block*. The current length of this field is defined to be exactly 64 bytes.

DES_key_block

Direction: Input/Output Type: String

The DES key information returned from a previous SET Block Compose service. The contents of the *DES_key_block* is the 64-byte DES internal key token (containing the DES key enciphered under the host master key). Your application program must not change the data in this string.

RSA_OAEP_block_length

Direction: Input/Output Type: Integer

The length of a block of storage to hold the *RSA-OAEP_block*. The length must be at least 128 bytes on input. The length value will be updated on exit with the actual length of the *RSA-OAEP_block*, which is exactly 128 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

RSA_OAEP_block

Direction: Output Type: String

The OAEP-formatted data block, encrypted under the RSA public key passed as *RSA_public_key_identifier*. When the OAEP-formatted data block is returned, it is left justified within the *RSA-OAEP_block* field if the input field length (*RSA-OAEP_block_length*) was greater than 128 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

chaining_vector

Direction: Input/Output	Type: String

An 18-byte field that ICSF uses as a system work area. Your application program must not change the data in this string. This field is ignored by this service, but must be specified.

DES_encrypted_data_block

Direction: Output	Type: String

The DES-encrypted data block (data passed in as *data_to_encrypt*). The length of the encrypted data is returned in *data_to_encrypt_length*. The *DES_encrypted_data_block* may be 8 bytes longer than the length of the *data_to_encrypt* because of padding added by this service.

Restrictions

- Caller must be task mode and must not be SRB mode.
- Not all CCA implementations support a key label as input in the *RSA_public_key_identifier* parameter. Some implementations may only support a key token.
- The *data_to_encrypt* and the *DES_encrypted_data_block* cannot overlap.
- NOCV keys must be installed in the CKDS to use SET block compose service on a CDMF-only system.

Usage Notes

- RACF will be invoked to check authorization to use the SET Block Compose service.
- The first time the SET Block Compose service is invoked to form an RSA-OAEP block and DES-encrypt data for communication between a specific source and destination (for example, between the merchant and payment gateway), do not specify the DES-ONLY keyword. A DES key will be generated by the service and returned in the key token contained in the *DES_key_block*. On subsequent calls to the Compose SET Block service for communication between the same source and destination, the DES key can be re-used. The caller of the service must supply the *DES_key_block*, the *DES_key_block_length*, the *data_to_encrypt*, the *data_to_encrypt_length*, and the rule-array keywords SET1.00 and DES-ONLY. You do not need to supply the block contents identifier, XDATA string and length, RSA-OAEP block and length, and RSA public key information, although you must still specify the parameters. For this invocation, the RSA-OAEP formatting is bypassed and only DES encryption is performed, using the supplied DES key.

SET Block Decompose (CSNDSBD)

Decomposes the RSA-OAEP block and the DES-encrypted data block of the SET protocol to provide unencrypted data back to the caller.

SET Block Decompose (CSNDSBD)

The SET block decompose callable service will route the request to a PCI Cryptographic Coprocessor for RSA-OAEP processing if there is a PCI Cryptographic Coprocessor available. The service has a preference for being processed on a PCI Cryptographic Coprocessor so that the symmetric key does not appear in the clear. If there is no PCI Cryptographic Coprocessor available, the request will be processed on the Cryptographic Coprocessor Feature, unless the *RSA_private_key_identifier* specifies a retained private key or a CRT form private key with private key section identifier of X'08', or unless the PINBLOCK rule array keyword was specified. These cases require a PCI Cryptographic Coprocessor, and the service will fail if no PCI Cryptographic Coprocessor is available.

Format

```
CALL CSNDSBD(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            RSA_OAEP_block_length,
            RSA_OAEP_block,
            DES_encrypted_data_block_length,
            DES_encrypted_data_block,
            initialization_vector,
            RSA_private_key_identifier_length,
            RSA_private_key_identifier,
            DES_key_block_length,
            DES_key_block,
            block_contents_identifier,
            XData_string_length,
            XData_string,
            chaining_vector,
            data_block,
            hash_block_length,
            hash_block)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” on page 335 lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” on page 335 lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be at least 1.

rule_array

Direction: Input Type: String

One keyword that provides control information to the callable service. The keyword indicates the block type. The keyword must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks.

Table 70. Keywords for SET Block Compose Control Information

Keyword	Meaning
Block Type (required)	
SET1.00	The structure of the RSA-OAEP encrypted block is defined by SET protocol.
Formatting Information (optional)	
DES-ONLY	DES decryption only is to be performed; no RSA-OAEP block decryption will be performed. (See Usage Notes.)
PINBLOCK	Specifies that the OAEP block will contain PIN information in the XDATA field, including an ISO-0 format PIN block. The *DES_key_block* must be 128 bytes in length and contain a IPINENC or OPINENC key. The PIN block will be encrypted under the PIN encrypting key. The PIN information and the encrypted PIN block are returned in the *XDATA_string* parameter.

RSA_OAEP_block_length

Direction: Input Type: Integer

The length of *RSA-OAEP_block* must be 128 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

RSA_OAEP_block

Direction: Input Type: String

The RSA-encrypted OAEP-formatted data block. This parameter is ignored if DES-ONLY is specified in the rule-array.

DES_encrypted_data_block_length

Direction: Input/Output | Type: Integer

The length in bytes of the DES-encrypted data block. The input length must be a multiple of 8 bytes. Updated on return to the length of the decrypted data returned in *data_block*. The maximum value of *DES_encrypted_data_block_length* is 32MB bytes.

DES_encrypted_data_block

Direction: Input | Type: String

The DES-encrypted data block. The data will be decrypted and passed back as *data_block*.

initialization_vector

Direction: Input | Type: String

An 8-byte string containing the initialization vector to be used for the cipher block chaining for the DES decryption of the data in the *DES_encrypted_data_block* parameter. You must use the same initialization vector that was used to perform the DES encryption of the data.

RSA_private_key_identifier_length

Direction: Input | Type: Integer

The length of the *RSA_private_key_identifier* field. The maximum size is 2500 bytes. This parameter is ignored if DES-ONLY is specified in the rule-array.

RSA_private_key_identifier

Direction: Input | Type: String

A key label of the RSA private key or an internal token of the RSA private key to be used to decipher the RSA-OAEP block passed in *RSA-OAEP_block*. The modulus bit length of the key must be 1024. This parameter is ignored if DES-ONLY is specified in the rule-array.

DES_key_block_length

Direction: Input/Output | Type: Integer

The length of the *DES_key_block*. The current length of this field may be 64 or 128 bytes. If rule array keyword PINBLOCK is specified, the length must be 128 bytes.

DES_key_block

Direction: Input/Output | Type: String

The *DES_key_block* contains either one or two DES internal key tokens. If only one token is specified on input, it contains either a null DES token (or binary zeroes) or (if DES-ONLY is specified) the DES key information returned from a previous SET Block Decompose service invocation. This is the 64-byte DES internal key token formed with the DES key which was retrieved from the RSA-OAEP block and enciphered under the host master key. Your application must not change this DES key information. If two tokens are specified in the *DES_key_block*, the first 64 bytes contain the DES token described above. The second 64 bytes, used when PINBLOCK is specified in the rule array, contain a DES internal token of the IPINENC or OPINENC key which is used to encrypt the PIN block returned to the caller in the XData_string parameter.

block_contents_identifier

Direction: Output	Type: String

A one-byte string, containing the binary value from the block contents (BC) field of the SET data block (DB). It indicates what data is carried in the actual data block (ADB) and the format of any extra data (*XData_string*). This parameter is ignored if DES-ONLY is specified in the rule-array.

XData_string_length

Direction: Input/Output	Type: Integer

The length of a string where the data contained within *XData_string* will be returned. The string must be at least 94 bytes in length. The value will be updated upon exit with the actual length of the returned *XData_string*. This parameter is ignored if DES-ONLY is specified in the rule-array.

XData_string

Direction: Output	Type: String

Extra-encrypted data contained within the OAEP-processed and RSA-encrypted block. The format is indicated by *block_contents_identifier*. The string is treated by ICSF as a string of hexadecimal digits. The service will always return the data from the beginning of the XDataString to the end of the SET DB block, a maximum of 94 bytes of data. The caller must examine the value returned in *block_contents_identifier* to determine the actual length of the XDataString. This parameter is ignored if DES-ONLY is specified in the rule-array.

chaining_vector

Direction: Input/Output	Type: String

An 18-byte field that ICSF uses as a system work area. Your application program must not change the data in this string. This field is ignored by this service, but must be specified.

data_block

Direction: Output	Type: String

The data that was decrypted (passed in as *DES_encrypted_data_block*). Any padding characters are removed.

hash_block_length

Direction: Input/Output	Type: Integer

The length in bytes of the SHA-1 hash returned in *hash_block*. On input, this parameter must be set to the length of the *hash_block* field. The length must be at least 20 bytes. On output, this field is updated to reflect the length of the SHA-1 hash returned in the *hash_block* field (exactly 20 bytes). This parameter is ignored if DES-ONLY is specified in the *rule_array* parameter.

hash_block

Direction: Output	Type: String

The SHA-1 hash extracted from the RSA-OAEP block. This parameter is ignored if DES-ONLY is specified in the *rule_array* parameter.

Restrictions

- Caller must be task mode and must not be SRB mode.
- Not all CCA implementations support a key label as input in the *RSA_private_key_identifier* parameter. Some implementations may only support a key token.
- The RSA private key used by this service must have been generated as a signature-only key.
- The *data_block* and the *DES_encrypted_data_block* cannot overlap.
- The ANSI system keys must be installed in the CKDS to use the SET block decompose service on a CDMF-only system.

Usage Notes

- RACF is invoked to check authorization to use the SET Block Decompose service.
- When the SET Block Decompose service is invoked without the DES-ONLY keyword, the DES key is retrieved from the RSA-OAEP block and returned in the key token contained in the *DES_key_block*. On subsequent calls to the SET Block Decompose service, a caller can re-use the DES key. The caller of the service must supply the *DES_key_block*, the *DES_key_block_length*, the *DES_encrypted_data_block*, the *DES_encrypted_data_block_length*, the initialization and chaining vectors, and the *rule_array* keywords SET1.00 and DES-ONLY. The RSA private key information, RSA-OAEP block and length, XData string and length, and hash block and length need not be supplied (although the parameters must still be specified). For this invocation, the decryption of the RSA-OAEP block is bypassed; only DES decryption is performed, using the supplied DES key.
- When the SET Block Decompose service is invoked with the PINBLOCK keyword, DES-ONLY may not also be specified. If both of these rule array keywords are specified, the service will fail with a Return code 8, Reason code 2016 (invalid rule array content). If PINBLOCK is specified and the *DES_key_block_length* field is not 128, the service will fail with Return code 8, Reason code 2790 (rule array keyword parameter mismatch).

VISA CVV Service Generate (CSNBCSG)

Use the VISA CVV Service Generate callable service to generate a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2. This service generates a CVV that is based upon the information that the *PAN_data*, the *expiration_date*, and the *service_code* parameters provide. The service uses the Key-A and the Key-B keys to cryptographically process this information. A key type of DATA is required for Key-A and Key-B. If the input values for Key-A and Key-B are not both single-length data keys, they will be routed to the PCI Cryptographic Coprocessor for processing. The PCI Cryptographic Coprocessor will allow Key-A and Key-B to be single-length MAC keys. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The CVV is returned in the 5-byte variable that the *CVV_value* parameter identifies. When you verify a CVV, compare the result to the value that the *CVV_value* supplies.

Format

```
CALL CSNBCSG(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             PAN_data,
             expiration_date,
             service_code,
             CVV_key_A_Identifier,
             CVV_key_B_Identifier,
             CVV_value)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Section Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Section Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The parameter *rule_array_count* must be 0, 1, or 2.

rule_array

Direction: Input	Type: String

Keywords that provide control information to the callable service. Each keyword is left-justified in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage.

Table 71. CVV Generate Rule Array Keywords

Keyword	Meaning
PAN data length (optional)	
PAN-13	Specifies that the length of the PAN data is 13 bytes. **PAN-13 is the default value.**
PAN-16	Specifies that the length of the PAN data is 16 bytes.
CVV length (optional)	
CVV-1	Specifies that the CVV is to be computed as one byte, followed by 4 blanks. **CVV-1 is the default value.**
CVV-2	Specifies that the CVV is to be computed as 2 bytes, followed by 3 blanks.
CVV-3	Specifies that the CVV is to be computed as 3 bytes, followed by 2 blanks.
CVV-4	Specifies that the CVV is to be computed as 4 bytes, followed by 1 blank.
CVV-5	Specifies that the CVV is to be computed as 5 bytes.

PAN_data

Direction: Input	Type: String

The *PAN_data* parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the **PAN-13** keyword is specified in the rule array, 13 characters are processed; if the **PAN-16** keyword is specified in the rule array, 16 characters are processed.

Even if you specify the **PAN-13** keyword, the server might copy 16 bytes to a work area. Therefore ensure that the verb can address 16 bytes of storage.

expiration_date

Direction: Input	Type: String

The *expiration_date* parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

service_code

Direction: Input	Type: String

The *service_code* parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of ’000’ is supported.

CVV_key_A_Identifier

Direction: Input/Output	Type: String

The *CVV_key_A_Identifier* parameter specifies an address that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key-A key that encrypts information in the CVV process.

CVV_key_B_Identifier

Direction: Input/Output	Type: String

The *CVV_key_B_Identifier* parameter specifies an address that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key-B key that decrypts information in the CVV process.

CVV_value

Direction: Output	Type: String

The *CVV_value* parameter specifies an address that points to the place in application data storage that will be used to store the computed 5-byte character output value.

Restriction

The CVV Generate verb is not supported on CDMF-only configurations.

VISA CVV Service Verify (CSNBCSV)

Use the VISA CVV service verify callable service to verify a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2. This service generates a CVV that is based upon the information that the *PAN_data*, the *expiration_date*, and the *service_code* parameters provide. The service uses the Key-A and the Key-B keys to cryptographically process this information. If the input values for Key-A and Key-B are not both single-length data keys, they will be routed to the PCI Cryptographic Coprocessor for processing. The PCI Cryptographic Coprocessor will allow Key-A and Key-B to be single-length MAC or MACVER keys. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The generated CVV is then compared to the value that the *CVV_value* supplies for verification.

Format

```
CALL CSNBCSV(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            PAN_data,
            expiration_date,
            service_code,
            CVV_key_A_Identifier,
            CVV_key_B_Identifier,
            CVV_value)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The parameter *rule_array_count* must be 0, 1, or 2.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service. Each keyword is left-justified in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage.

Table 72. CVV Verify Rule Array Keywords

Keyword	Meaning
PAN data length (optional)	
PAN-13	Specifies that the length of the PAN data is 13 bytes. **PAN-13 is the default value.**
PAN-16	Specifies that the length of the PAN data is 16 bytes.
CVV length (optional)	
CVV-1	Specifies that the CVV is to be computed as one byte, followed by 4 blanks. **CVV-1 is the default value.**
CVV-2	Specifies that the CVV is to be computed as 2 bytes, followed by 3 blanks.
CVV-3	Specifies that the CVV is to be computed as 3 bytes, followed by 2 blanks.
CVV-4	Specifies that the CVV is to be computed as 4 bytes, followed by 1 blank.
CVV-5	Specifies that the CVV is to be computed as 5 bytes.

PAN_data

Direction: Input Type: String

The *PAN_data* parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the **PAN-13** keyword is specified in the rule array, 13 characters are processed; if the **PAN-16** keyword is specified in the rule array, 16 characters are processed.

Even if you specify the **PAN-13** keyword, the server might copy 16 bytes to a work area. Therefore ensure that the verb can address 16 bytes of storage.

expiration_date

Direction: Input Type: String

The *expiration_date* parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

service_code

Direction: Input Type: String

The *service_code* parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of ’000’ is supported.

CVV_key_A_Identifier

Direction: Input/Output   Type: String

The *CVV_key_A_Identifier* parameter specifies an address that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key-A key that encrypts information in the CVV process.

CVV_key_B_Identifier

Direction: Input/Output   Type: String

The *CVV_key_B_Identifier* parameter specifies an address that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key-B key that decrypts information in the CVV process.

CVV_value

Direction: Input   Type: String

The *CVV_value* parameter specifies an address that contains the CVV value which will be compared to the computed CVV value.

Restrictions

The CVV Verify verb is not supported on CDMF-only configurations.

Chapter 8. Using Digital Signatures

This chapter describes the PKA callable services that support using digital signatures to authenticate messages.

- "Digital Signature Generate (CSNDDSG)"
- "Digital Signature Verify (CSNDDSV)" on page 269

Digital Signature Generate (CSNDDSG)

Use the digital signature generate callable service to generate a digital signature using a PKA private key. The digital signature generate callable service may use either the RSA or DSS private key, depending on the algorithm you are using. The RSA private key must be valid for signature usage. This service supports the following methods:

- ANSI X9.30 (DSS)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

Note: The maximum signature length is 256 bytes (2048 bits).

The input text should have been previously hashed using either the one-way hash generate callable service or the MDC generation callable service. If the signature formatting algorithm specifies ANSI X9.31, you must specify the hash algorithm used to hash the text (SHA-1 or RPMD-160).

If the *PKA_private_key_identifier* specifies an RSA private key, you select the method of formatting the text through the *rule_array* parameter. If the *PKA_private_key_identifier* specifies a DSS private key, the DSS signature generated is according to ANSI X9.30. For DSS, the signature is generated on a 20-byte hash created from SHA-1 algorithm.

The digital signature generate callable service examines the RSA key specified in the *PKA_private_key_identifier* to determine how to route the request. If the modulus bit length is less than 512 bits, or if the key is a modulus-exponent form private key with a private section ID of X'02', ICSF routes the request to the Cryptographic Coprocessor Feature. If the key is a X'08' form CRT private key, or a retained private key, the service routes the request to a PCI Cryptographic Coprocessor. In the case of a retained key, the service routes the request to the specific PCI Cryptographic Coprocessor in which the key is retained. If the key is a modulus-exponent form private key with a private section ID of X'06', the service routes the request as follows:

- If the key use bits indicate signature use only, the digital signature generate service routes the request to either a Cryptographic Coprocessor Feature or a PCI Cryptographic Coprocessor depending upon availability. If there is no PCI Cryptographic Coprocessor online, the request is routed to a Cryptographic Coprocessor Feature.
- If the key use bits indicate key-management use is allowed and the KMMK is equal to the SMK on the Cryptographic Coprocessor Feature, the digital signature generate service routes the request to either a Cryptographic Coprocessor Feature or a PCI Cryptographic Coprocessor depending upon

availability. If there is no PCI Cryptographic Coprocessor online, the request is routed to a Cryptographic Coprocessor Feature.

- If the key use bits indicate key-management use is allowed and the KMMK is not equal to the SMK on the Cryptographic Coprocessor Feature, the request must be processed on a PCI Cryptographic Coprocessor. If there is no PCI Cryptographic Coprocessor online, the request will fail and issue a return and reason code.

Note: For PKCS the message digest and the message-digest algorithm identifier are combined into an ASN.1 value of type DigestInfo, which is BER-encoded to give an octet string D (see Table 73). D is the text string supplied in the *hash* variable.

Format

```
CALL CSNDDSG(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            PKA_private_key_identifier_length,
            PKA_private_key_identifier,
            hash_length,
            hash,
            signature_field_length,
            signature_bit_length,
            signature_field)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value may be 0 1, or 2.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service. A keyword specifies the method for calculating the RSA digital signature. Table 73 lists the keywords. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

Table 73. Keywords for Digital Signature Generate Control Information.. Valid only for RSA key types.

Keyword	Meaning
Digital Signature Formatting Method (optional)	
ISO-9796	Calculate the digital signature on the *hash* according to ISO-9796-1. Any hash method is allowed. This is the default.
PKCS-1.0	Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must have been hashed and BER-encoded before input to this service.
PKCS-1.1	Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must have been hashed and BER-encoded before input to this service.
ZERO-PAD	Format the hash by padding it on the left with binary zeros to the length of the RSA key modulus. Any supported hash function is allowed.
X9.31	Format according to the ANSI X9.31 standard. The input text must have been previously hashed with one of the hash algorithms specified below.
Hash Method Specification: Required with X9.31	
RPMD-160	Hash the input text using the RIPEMD-160 hash method.
SHA-1	Hash the input text using the SHA-1 hash method.

PKA_private_key_identifier_length

Direction: Input	Type: Integer

The length of the *PKA_private_key_identifier* field. The maximum size is 2500 bytes.

PKA_private_key_identifier

Direction: Input	Type: String

An internal token or label of the PKA private key or Retained key. If the signature format is X9.31, the modulus of the RSA key must have a length of at least 1024 bits.

hash_length

Direction: Input	Type: Integer

The length of the *hash* parameter in bytes. It must be the exact length of the text to sign. The maximum size is 256 bytes. Beginning with OS/390 V2 R9 ICSF, if you specify ZERO-PAD in the *rule_array parameter*, the input hash length is limited to 32 bytes (256 bits). APAR OW48511 (for OS/390 V2 R9 and OS/390 V2 R10) changes the hash length limit to 256 bytes when ZERO-PAD is specified for signature use only keys. It also increases the hash length limit for all other keys when ZERO-PAD is specified to 36 bytes.

hash

Direction: Input	Type: String

The application-supplied text on which to generate the signature. The input text must have been previously hashed, and for PKCS formatting, it must be BER-encoded as previously described. For X9.31, the hash algorithms must have been either SHA-1 or RIPEMD-160. See the *rule_array* parameter for more information.

signature_field_length

Direction: Input/Output	Type: Integer

The length in bytes of the *signature_field* to contain the generated digital signature.

Note: For RSA, this must be at least the RSA modulus size (rounded up to a multiple of 32 bytes for the X9.31 signature format, or one byte for all other signature formats). For DSS, this must be at least 40 bytes. For RSA and DSS, this field is updated with the minimum byte length of the digital signature. The maximum size is 256 bytes.

signature_bit_length

Direction: Output	Type: Integer

The bit length of the digital signature generated. For ISO-9796 this is 1 less than the modulus length. For other RSA processing methods, this is the modulus length. For DSS, this is 320.

signature_field

Direction: Output	Type: String

The digital signature generated is returned in this field. The digital signature is in the low-order bits (right-justified) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-justified within the *signature_field*. Any unused bytes to the right are undefined.

Restrictions

Although ISO-9796 does not require the input hash to be an integral number of bytes in length, this service requires you to specify the *hash_length* in bytes.

The caller must be in task mode and not in SRB mode.

X9.31 requires the RSA token to have a modulus bit length of at least 1024 bits and the length must also be a multiple of 256 bits (or 32 bytes).

Beginning with OS/390 V2 R9 ICSF, if you specify ZERO-PAD in the *rule_array parameter*, the input hash length is limited to 32 bytes (256 bits). APAR OW48511 (for OS/390 V2 R9 and OS/390 V2 R10) changes the hash length limit to 256 bytes when ZERO-PAD is specified for signature use only keys. It also increases the hash length limit for all other keys when ZERO-PAD is specified to 36 bytes.

Digital Signature Verify (CSNDDSV)

Use the digital signature verify callable service to verify a digital signature using a PKA public key. The digital signature verify callable service can use the RSA or DSS public key, depending on the digital signature algorithm used to generate the signature. This service supports the following methods:

- ANSI X9.30 (DSS)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

Input text should have been previously hashed. You can use either the one-way hash generate callable service or the MDC generation callable service.

This service routes requests to the Cryptographic Coprocessor Feature.

Note: The maximum signature length is 256 bytes (2048 bits).

Format

```
CALL CSNDDSV(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            PKA_public_key_identifier_length,
            PKA_public_key_identifier,
            hash_length,
            hash,
            signature_field_length,
            signature_field)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be 0 or 1.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service. A keyword specifies the method to use to verify the RSA digital signature. Table 74 lists the keywords. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

Table 74. Keywords for Digital Signature Verify Control Information. Valid Only for RSA Key Types.

Keyword	Meaning
ISO-9796-1	Verify the digital signature on the hash according to ISO-9796-1. Any hash method is allowed. This is the default.
PKCS-1.0	Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must specify BER encoded hash text.
PKCS-1.1	Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must specify BER encoded hash text.
ZERO-PAD	Format the hash by padding it on the left with binary zeros to the length of the PKA key modulus. Any supported hash function is allowed.
X9.31	Format according to ANSI X9.31 standard.

PKA_public_key_identifier_length

Direction: Input Type: Integer

The length of the *PKA_public_key_identifier* field containing the public key token or label. The maximum size is 2500 bytes.

PKA_public_key_identifier

Direction: Input Type: String

A token or label of the PKA public key.

hash_length

Direction: Input Type: Integer

The length of the *hash* parameter in bytes. It must be the exact length of the text that was signed. The maximum size is 256 bytes.

hash

Direction: Input Type: String

The application-supplied text on which the supplied signature was generated. The text must have been previously hashed and, for PKCS formatting, BER-encoded as previously described.

signature_field_length

Direction: Input	Type: Integer

The length in bytes of the *signature_field* parameter. The maximum size is 256 bytes.

signature_field

Direction: Input	Type: String

This field contains the digital signature to verify. The digital signature is in the low-order bits (right-justified) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-justified within the *signature_field*.

Restrictions

The ability to recover a message from a signature (which ISO-9796 allows but does not require) is **not** supported.

The exponent of the RSA public key must be odd.

Although ISO-9796 does not require the input hash to be an integral number of bytes in length, this service requires you to specify the *hash_length* in bytes.

The caller must be in task mode and not in SRB mode.

X9.31 requires the RSA token to have a modulus bit length of at least 1024 bits and the length must also be a multiple of 256 bits (or 32 bytes).

Usage Note

For DSS if r=0 or s=0 then verification always fails. The DSS digital signature is of the form r || s, each 20 bytes.

Chapter 9. Managing PKA Cryptographic Keys

This chapter describes the callable services that generate and manage PKA keys.

PKA Key Generate (CSNDPKG)

Use the PKA key generate callable service to generate the following PKA keys:

- PKA internal tokens for use with the DSS algorithm in the digital signature services
- RSA keys for use on the Cryptographic Coprocessor Feature or PCI Cryptographic Coprocessor

Input to the PKA key generate callable service is either a skeleton key token that has been built by the PKA key token build service or a valid internal token. In the case of a valid internal token, PKG will generate a key with the same modulus length and the same exponent. The service examines the skeleton token and routes the generation request to the appropriate cryptographic processor. If the skeleton is a DSS key token, processing takes place on the Cryptographic Coprocessor Feature. If the skeleton is an RSA key token, processing takes place on the PCI Cryptographic Coprocessor.

DSS key generation requires the following information in the input skeleton token:

- Size of modulus p in bits
- Prime modulus p
- Prime divisor q
- Public generator g
- Optionally, the private key name

DSS standards define restrictions on p, q, and g. (Refer to the Federal Information Processing Standard (FIPS) Publication 186 for DSS standards.) This callable service does not verify all of these restrictions. If you do not follow these restrictions, the keys you generate may not be valid DSS keys. The PKA Key Token Build service or an existing internal or external PKA DSS token can generate the input skeleton token, but all of the preceding must be provided. You can extract the DSS public key token from the internal private key token by calling the PKA public key extract callable service.

RSA key generation requires the following information in the input skeleton token:

- Size of the modulus in bits. The modulus for modulus-exponent form keys is between 512 and 1024. The CRT modulus is between 512 and 2048.

RSA key generation has the following restrictions: For modulus-exponent, there are restrictions on modulus, public exponent, and private exponent. For CRT, there are restrictions on dp, dq, U, and public exponent. See the Key value structure in "PKA Key Token Build (CSNDPKB)" on page 279 for a summary of restrictions.

PKA Key Generate (CSNDPKG)

Note: The Transaction Security System PKA96 PKA key generate verb supports RSA key generation only; it does not support DSS key generation.

Format

```
CALL CSNDPKG(
           return_code,
           reason_code,
           exit_data_length,
           exit_data,
           rule_array_count,
           rule_array,
           regeneration_data_length,
           regeneration_data,
           skeleton_key_identifier_length,
           skeleton_key_identifier,
           transport_key_identifier,
           generated_key_token_length,
           generated_key_token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. Value may be 1 or 2.

rule_array

Direction: Input Type: String

A keyword that provides control information to the callable service. See Table 75 for a list. A keyword is left-justified in an 8-byte field and padded on the right with blanks.

Table 75. Keyword for PKA Key Generate Rule Array

Keyword	Meaning
Private Key Encryption (required)	
CLEAR	Return the private key in clear text. The private key in clear text is an external token. Only valid for RSA keys.
MASTER	Encipher the private key under the master key.
RETAIN	Retain the private key within the PCI Cryptographic Coprocessor for additional security. Only valid for RSA keys.
XPORT	Encipher the private key under the *transport_key_identifier*. Only valid for RSA keys.
Options (optional)	
CLONE	Mark a generated and retained private key as usable in cryptographic engine cloning process. This keyword is supported only if RETAIN is also specified. Only valid for RSA keys.

regeneration_data_length

Direction: Input Type: Integer

The value must be 0 for DSS tokens. For RSA tokens, the regeneration_data_length can be non-zero. If it is non-zero, it must be between 8 and 256 bytes inclusive.

regeneration_data

Direction: Input Type: String

This field points to a string variable containing a string used as the basis for creating a particular public-private key pair in a repeatable manner.

skeleton_key_identifier_length

Direction: Input Type: Integer

The length of the *skeleton_key_identifier* parameter in bytes. The maximum allowed value is 2500 bytes.

skeleton_key_identifier

Direction: Input Type: String

The application-supplied skeleton key token generated by PKA key token build or label of the token that contains the required network quantities for DSS key generation, or the required modulus length and public exponent for RSA key generation.

transport_key_identifier

Direction: Input	Type: String

A 64-byte field to contain a DES key identifier. This field must be binary zeros, unless the XPORT rule is specified. For XPORT rule, this is an IMPORTER or EXPORTER key or the label of an IMPORTER or EXPORTER key that is used to encrypt the generated key. If you specify a label, it must resolve uniquely to either an IMPORTER or EXPORTER key. Only valid for RSA keys.

generated_key_token_length

Direction: Input/Output	Type: Integer

The length of the generated key token. The field is checked to ensure it is at least equal to the token being returned. The maximum size is 2500 bytes. On output, this field is updated with the actual token length.

generated_key_token

Direction: Input/Output	Type: String

The internal token or label of the generated DSS or RSA key. The label can be that of a retained key. Checks are made to ensure that a retained key is not overlayed in PKDS. If the label is that of a retained key, the private name in the token must match the label name. If a label is specified in the *generated_key_token* field, the *generated_key_token_length* returned to the application will be the same as the input length. If RETAIN was specified, but the *generated_key_token* was not specified as a label, the generated key length returned to the application will be zero (the key was retained in the PCI Cryptographic Coprocessor). If the record already exists in the PKDS with the same label as the one specified as the *generated_key_token*, the record will be overwritten with the newly generated key token (unless the PKDS record is an existing retained private key, in which case it cannot be overwritten). If there is no existing PKDS record with this label in the case of generating a retained key, a record will be created. For generation of a non-retained key, if a label is specified in the *generated_key_token* field, a record must already exist in the PKDS with this same label or the service will fail.

Restriction

The caller must be in task mode and not in SRB mode.

Usage Note

When a Retained key is created, ICSF records this event in a type 82 SMF record with a subtype of 15.

SAF will be invoked to check authorization to use any key labels specified as input.

| Beginning with OS/390 V2 R9 ICSF, CSFDPKG now supports writing the
| *generated_key* directly to the PKDS. This means that the *generated_key_token* field

is now an INPUT as well as an OUTPUT field. If a PKDS label name is not being supplied, than a value less than a blank X'40' must be supplied in the first byte of the parameter or else the service fails with a return code 8, reason code 11000.

PKA Key Import (CSNDPKI)

This service imports an external PKA private key token. (This consists of a PKA private key and public key.) The secret values of the key may be clear or encrypted under a limited-authority DES importer key.

This service can also import a clear PKA key. The PKA key token build service creates a clear PKA key token.

Output of this service is an ICSF internal token of the RSA or DSS private key.

ICSF examines the key token supplied in the *source_key_identifier* to determine where to route the request. If the *source_key_identifier* contains an RSA private key with a modulus length of at least 512 bits, ICSF routes the PKA key import request to a PCI Cryptographic Coprocessor. If no PCI Cryptographic Coprocessor is online, or if the *source_key_identifier* contains either an RSA private key with a modulus length less than 512 bits or a DSS private key, ICSF routes the PKA key import request to the Cryptographic Coprocessor Feature. An RSA modulus-exponent form token imported on the PCI Cryptographic Coprocessor results in an X'06' format, while a token imported on a Cryptographic Coprocessor Feature will result in a X'02' format. If no PCI Cryptographic Coprocessor is online and the *source_key_identifier* is an RSA-CRT token, the request fails.

Format

```
CALL CSNDPKI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            source_key_identifier_length,
            source_key_identifier,
            importer_key_identifier,
            target_key_identifier_length,
            target_key_identifier)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. This must be 0.

rule_array

Direction: Input	Type: String

Reserved field. This field is not used, but you must specify it.

source_key_identifier_length

Direction: Input	Type: Integer

The length of the *source_key_identifier* parameter. The maximum size is 2500 bytes.

source_key_identifier

Direction: Input	Type: String

The external token or label of a PKA private key. This cannot be the label of a retained private key. This is the output of the PKA key generate (CSNDPKG) callable service or the PKA key token build (CSNDPKB) callable service. If encrypted, it was created on another platform.

importer_key_identifier

Direction: Input/Output	Type: String

A DES internal token or the label of an IMP-PKA key. This is a limited authority key-encrypting key. It is ignored for clear tokens.

target_key_identifier_length

Direction: Input/Output	Type: Integer

The length of the *target_key_identifier* parameter. The maximum size is 2500 bytes.

target_key_identifier

Direction: Input/Output	Type: String

This field contains the internal token or label of the imported PKA private key. If a label is specified on input, a PKDS record with this label must exist. The PKDS record with this label will be overwritten with imported key unless the existing record is a retained key. If the record is a retained key, the import will fail. A retained key record cannot be overwritten. If no label is specified on input, this field should be set to binary zeroes on input.

Restrictions

This service imports RSA keys of up to 2048 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key may be successfully imported but fail when used if the limits are exceeded.

The *importer_key_identifier* is a limited-authority key-encrypting key.

The caller must be in task mode and not in SRB mode.

CRT form tokens with a private section ID of X'05' cannot be imported into ICSF.

Usage Notes

This service imports keys of any modulus size up to 2048 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key may be successfully imported but fail when used if the limits are exceeded.

Beginning with OS/390 V2 R9 ICSF, CSFDPKI now supports writing the *target_key_identifier* directly to the PKDS. This means that the *target_key_identifier* field is now an INPUT as well as an OUTPUT field.

If a PKDS label name is not being supplied, then a value less than a blank (X'40') must be supplied in the first byte of the parameter or else the service fails with return code 8, reason code 11000.

PKA Key Token Build (CSNDPKB)

Use this utility to build external PKA key tokens containing unenciphered private RSA or DSS keys. You can use this token as input to the PKA key import service to obtain an operational internal token containing an enciphered private key. This service builds a skeleton token you can use as input to the PKA key generate callable service (see Table 75 on page 275). You can also input to this service a clear unenciphered public RSA or DSS key and return the public key in a token format that other ICSF PKA services can use directly.

You can also use this service to build a key token for an RSA private key in optimized Chinese Remainder Theorem (CRT) form.

DSS key generation requires the following information in the input skeleton token:

- Size of modulus p in bits
- Prime modulus p
- Prime divisor q
- Public generator g
- Optionally, the private key name

Note: DSS standards define restrictions on the prime modulus p, prime divisor q, and public generator g. (Refer to the Federal Information Processing Standard (FIPS) Publication 186 for DSS standards.) This callable service does not verify all of these restrictions. If you do not follow the restrictions, the keys you generate may not be valid DSS keys.

Format

```
CALL CSNDPKB(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_value_structure_length,
            key_value_structure,
            private_key_name_length,
            private_key_name,
            reserved_1_length,
            reserved_1,
            reserved_2_length,
            reserved_2,
            reserved_3_length,
            reserved_3,
            reserved_4_length,
            reserved_4,
            reserved_5_length,
            reserved_5,
            key_token_length,
            key_token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Ignored Type: Integer

Reserved field.

exit_data

Direction: Input/Output Type: String

Reserved field.

rule_array_count

Direction: Input Type: Integer

The number of keywords you supplied in the *rule_array* parameter. Value must be 1 or 2.

rule_array

Direction: Input Type: String

One or two keywords that provide control information to the callable service. Table 76 lists the keywords. The keywords must be in 8 to 16 bytes of contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks.

Table 76. Keywords for PKA Key Token Build Control Information

Keyword	Meaning
Key Type (required)	
DSS-PRIV	This keyword indicates building a key token containing both public and private DSS key information. The parameter *key_value_structure* identifies the input key values, if supplied.
DSS-PUBL	This keyword indicates building a key token containing public DSS key information. The parameter *key_value_structure* identifies the input key values, if supplied.
RSA-CRT	This keyword indicates building a token containing an RSA private key in the optimized Chinese Remainder Theorem (CRT) form. The parameter *key_value_structure* identifies the input key values, if supplied.
RSA-PRIV	This keyword indicates building a token containing both public and private RSA key information. The parameter *key_value_structure* identifies the input key values, if supplied.
RSA-PUBL	This keyword indicates building a token containing public RSA key information. The parameter *key_value_structure* identifies the input values, if supplied.

Table 76. Keywords for PKA Key Token Build Control Information (continued)

Keyword	Meaning
Key Usage Control (optional)	
KEY-MGMT	Indicates that an RSA private key can be used in both the symmetric key import and the digital signature generate callable services.
KM-ONLY	Indicates that an RSA private key can be used only in symmetric key distribution.
SIG-ONLY	Indicates that an RSA private key cannot be used in symmetric key distribution. This is the default. Note that for DSS-PRIV the keyword is allowed but extraneous; DSS keys are defined only for digital signature.

key_value_structure_length

Direction: Input	Type: Integer

This is a segment of contiguous storage containing a variable number of input clear key values. The length depends on the key type parameter in the rule array and on the actual values input. The length is in bytes.

Table 77. Key Value Structure Length Maximum Values for Key Types

Key Type	Key Value Structure Maximum Value
DSS-PRIV	436
DSS-PUBL	416
RSA-CRT	2500
RSA-PRIV	648
RSA-PUBL	520

key_value_structure

Direction: Input	Type: String

This is a segment of contiguous storage containing a variable number of input clear key values and the lengths of these values in bits or bytes, as specified. The structure elements are ordered, of variable length, and the input key values must be right-justified within their respective structure elements and padded on the left with binary zeros. Table 78 defines the structure and contents as a function of key type.

Table 78. Key Value Structure Elements for PKA Key Token Build

Offset	Length (bytes)	Description
Key Value Structure (Optimized RSA, Chinese Remainder Theorem form, RSA-CRT)		
000	002	Modulus length in bits (512 to 2048). This is required.

Table 78. Key Value Structure Elements for PKA Key Token Build (continued)

Offset	Length (bytes)	Description
002	002	Modulus field length in bytes, “nnn.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. This value must not exceed 256.
004	002	Public exponent field length in bytes, “eee.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service.
006	002	Reserved, binary zero.
008	002	Length of the prime number, p, in bytes, “ppp.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. Maximum size of p + q is 256 bytes.
010	002	Length of the prime number, q, in bytes, “qqq.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. Maximum size of p + q is 256 bytes.
012	002	Length of d_p, in bytes, “rrr.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. Maximum size of $d_p + d_q$ is 256 bytes.
014	002	Length of d_q, in bytes, “sss.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. Maximum size of $d_p + d_q$ is 256 bytes.
016	002	Length of U, in bytes, “uuu.” This value can be zero if the key token is used as a *skeleton_key_token* in the PKA key generate callable service. Maximum size of U is 256 bytes.
018	nnn	Modulus, n.

Table 78. Key Value Structure Elements for PKA Key Token Build (continued)

Offset	Length (bytes)	Description
018 + nnn	eee	Public exponent, e. This is an integer such that 1<e<n. e must be odd. When you are building a *skeleton_key_token* to control the generation of an RSA key pair, the public key exponent can be one of the following values: 3, 65537 (2^{16} + 1), or 0 to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is 0.
018 + nnn + eee	ppp	Prime number, p.
018 + nnn + eee + ppp	qqq	Prime number, q.
018 + nnn + eee + ppp + qqq	rrr	d_p = d mod(p-1).
018 + nnn + eee + ppp + qqq + rrr	sss	d_q = d mod(q-1).
018 + nnn + eee + ppp + qqq + rrr + sss	uuu	U = q^{-1}mod(p).
Key Value Structure (RSA Private or RSA Public)		
000	002	Modulus length in bits. This is required. When building a skeleton token, the modulus length in bits must be greater than or equal to 512 bits.
002	002	Modulus field length in bytes, “XXX”. This value can be zero if you are using the key token as a skeleton in the PKA key generate verb. This value must not exceed 256 when the RSA-PUBL keyword is used, and must not exceed 128 when the RSA-PRIV keyword is used. This service can build a key token for a public RSA key with a 2048-bit modulus length, or it can build a key token for a 1024-bit modulus length private key.

Table 78. Key Value Structure Elements for PKA Key Token Build (continued)

Offset	Length (bytes)	Description
004	002	Public exponent field length in bytes, “YYY”. This value must not exceed 256 when the RSA-PUBL keyword is used, and must not exceed 128 when the RSA-PRIV keyword is used. This value can be zero if you are using the key token as a skeleton token in the PKA key generate verb. In this case, a random exponent is generated. To obtain a fixed, predetermined public key exponent, you can supply this field and the public exponent as input to the PKA key generate verb.
006	002	Private exponent field length in bytes, “ZZZ”. This field can be zero, indicating that private key information is not provided. This value must not exceed 128 bytes. This value can be zero if you are using the key token as a skeleton token in the PKA key generate verb.
008	XXX	Modulus, n. This is an integer such that $1<n<2^{2048}$. The n is the product of p and q for primes p and q.
008 + XXX	YYY	RSA public exponent e, which is an odd integer where $1<e<n$. You can supply this value in a skeleton to generate an RSA private key with a predetermined public exponent value.
008 + XXX + YYY	ZZZ	RSA secret exponent d. This is an integer such that $1<d<n$. The value of d is e^{-1} mod(p-1)(q-1); the You need not specify this value if you specify RSA-PUBL in the rule array.
Key Value Structure (DSS Private or DSS Public)		
000	002	Modulus length in bits. This is required.
002	002	Prime modulus field length in bytes, “XXX”. You can supply this as a network quantity to the ICSF PKA key generate callable service, which uses the quantity to generate DSS keys. The maximum allowed value is 128.

Table 78. Key Value Structure Elements for PKA Key Token Build (continued)

Offset	Length (bytes)	Description
004	002	Prime divisor field length in bytes, “YYY”. You can supply this as a network quantity to the ICSF PKA key generate callable service, which uses the quantity to generate DSS keys. The allowed values are 0 or 20 bytes.
006	002	Public generator field length in bytes, “ZZZ”. You can supply this in a skeleton token as a network quantity to the ICSF PKA key generate callable service, which uses the quantity to generate DSS keys. The maximum allowed value is 128 bytes and is exactly the same length as the prime modulus.
008	002	Public key field length in bytes, “AAA”. This field can be zero, indicating that the ICSF PKA key generate callable service generates a value at random from supplied or generated network quantities. The maximum allowed value is 128 bytes and is exactly the same length as the prime modulus.
010	002	Secret key field length in bytes, “BBB”. This field can be zero, indicating that the ICSF PKA key generate callable service generates a value at random from supplied or generated network quantities. The allowed values are 0 or 20 bytes.
012	XXX	DSS prime modulus p. This is an integer such that $2^{L-1}<p<2^{L}$. The p must be prime. You can supply this value in a skeleton token as a network quantity; it is used in the algorithm that generates DSS keys.
012 + XXX	YYY	DSS prime divisor q. This is an integer that is a prime divisor of p-1 and $2^{159}<q<2^{160}$. You can supply this value in a skeleton token as a network quantity; it is used in the algorithm that generates DSS keys.
012 + XXX+ YYY	ZZZ	DSS public generator g. This is an integer such that $1<g<p$. You can supply this value in a skeleton token as a network quantity; it is used in the algorithm that generates DSS keys.
012 + XXX+ YYY+ ZZZ	AAA	DSS public key y. This is an integer such that $y = g^{x} \bmod p$.

Table 78. Key Value Structure Elements for PKA Key Token Build (continued)

Offset	Length (bytes)	Description
012 + XXX+ YYY+ ZZZ+ AAA	BBB	DSS secret private key x. This is an integer such that 0<x<q. The x is random. You need not supply this value if you specify DSS-PUBL in the rule array.

Notes:

1. All length fields are in binary.
2. All binary fields (exponent, lengths, modulus, and so on) are stored with the high-order byte field first. This integer number is right-justified within the key structure element field.
3. You must supply all values in the structure to create a token containing an RSA or DSS private key for input to the PKA key import service.

private_key_name_length

Direction: Input Type: Integer

The length can be 0 or 64.

private_key_name

Direction: Input Type: EBCDIC character

This field contains the name of a private key. The name must conform to ICSF label syntax rules. That is, allowed characters are alphanumeric, national (@,#,$) or period (.). The first character must be alphabetic or national. The name is folded to upper case and converted to ASCII characters. ASCII is the permanent form of the name because the name should be independent of the platform. The name is then cryptographically coupled with clear private key data before encryption of the private key. Because of this coupling, the name can never change after the key token is imported. The parameter is valid only with key type RSA-CRT.

reserved_1_length

Direction: Input Type: Integer.

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_1

Direction: Input Type: String

The *reserved_1* parameter identifies a string that is reserved. The service ignores it.

reserved_2_length

Direction: Input Type: Integer.

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_2

Direction: Input	Type: String

The *reserved_2* parameter identifies a string that is reserved. The service ignores it.

reserved_3_length

Direction: Input	Type: Integer.

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_3

Direction: Input	Type: String

The *reserved_3* parameter identifies a string that is reserved. The service ignores it.

reserved_4_length

Direction: Input	Type: Integer.

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_4

Direction: Input	Type: String

The *reserved_4* parameter identifies a string that is reserved. The service ignores it.

reserved_5_length

Direction: Input	Type: Integer.

Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_5

Direction: Input	Type: String

The *reserved_5* parameter identifies a string that is reserved. The service ignores it.

key_token_length

Direction: Input/Output	Type: Integer

Length of the returned key token. The service checks the field to ensure it is at least equal to the size of the token to return. On return from this service, this field is updated with the exact length of the *key_token* created. On input, a size of 1024 bytes is sufficient to contain the largest *key_token* created.

key_token

Direction: Output   Type: String

The returned key token containing an unenciphered private or public key. The private key is in an external form that can be exchanged with different Common Cryptographic Architecture (CCA) PKA systems. You can use the public key token directly in appropriate ICSF signature verification or key management services.

Usage Note

If you are building a skeleton for use in a PKA Key Generate request to generate a retained PKA private key, you must build a private key name section in the skeleton token.

PKA Key Token Change (CSNDKTC)

The PKA Key Token Change callable service changes PKA key tokens (RSA and DSS) from encipherment under the old PCI Cryptographic Coprocessor Asymmetric-Keys Master Key to encipherment under the current PCI Cryptographic Coprocessor Asymmetric-Keys Master Key. This service only changes Private Internal PKA Key Tokens. PKA private keys encrypted under the Key Management Master Key (KMMK) cannot be reenciphered using this service unless the KMMK has the same value as the Signature Master Key (SMK).

ICSF routes the PKA Key Token Change request to a PCI Cryptographic Coprocessor for processing. If no PCI Cryptographic Coprocessor is online, the request fails.

Format

```
CALL CSNDKTC(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_identifier_length,
            key_identifier )
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be 1.

rule_array

Direction: Input Type: Character String

The process rule for the callable service. The keyword must be in 8 bytes of contiguous storage, left-justified and padded on the right with blanks.

Table 79. Rule Array Keywords for PKA Key Token Change (Required)

Keyword	Meaning
RTCMK	Changes the PKA key from encipherment with the old master key to encipherment with the current master key.

key_identifier_length

Direction: Input Type: Integer

The length of the *key_identifier* parameter. The maximum size is 2500 bytes.

key_identifier

Direction: Input/Output Type: String

An internal RSA or DSS private key token.

Usage Note

PKA callable services must be enabled to use the PKA Key Token Change callable service.

PKA Public Key Extract (CSNDPKX)

Use the PKA public key extract callable service to extract a PKA public key token from a supplied PKA internal or external private key token. This service performs no cryptographic verification of the PKA private token. You can verify the private token by using it in a service such as digital signature generate.

Format

```
CALL CSNDPKX(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             source_key_indentifier_length,
             source_key_identifier,
             target_public_key_token_length,
             target_public_key_token)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Ignored   Type: Integer

Reserved field.

exit_data

Direction: Ignored   Type: String

Reserved field.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. The value must be 0.

rule_array

Direction: Input	Type: String

Reserved field. This field is not used, but you must specify it.

source_key_identifier_length

Direction: Input	Type: integer

The length of the *source_key_identifier* parameter. The maximum size is 2500 bytes.

source_key_identifier

Direction: Input/output	Type: string

The internal or external token of a PKA private key or the label of a PKA private key. This can be the input or output from PKA key import or from PKA key generate.

This service supports the RSA private key token formats supported on the PCI Cryptographic Coprocessor. If the *source_key_identifier* specifies a label for a private key that has been retained within a PCI Cryptographic Coprocessor, this service extracts only the public key section of the token.

target_public_key_token_length

Direction: Input/Output	Type: Integer

The length of the *target_public_key_token* parameter. The maximum size is 2500 bytes. On output, this field will be updated with the actual byte length of the *target_public_key_token*.

target_public_key_token

Direction: Output	Type: String

This field contains the token of the extracted PKA public key.

Restriction

The caller must be in task mode and not in SRB mode.

Usage Notes

This service extracts the public key from the internal or external form of a private key. However, it does not check the cryptographic validity of the private token.

Beginning with OS/390 V2 R9 ICSF, this service must be in task mode, not SRB mode. It was also enhanced to support PKDS labels as well as tokens. This requires a change to the stub module CSNDPKX. Existing applications that have been link edited with the old stub module will still run without change. Access to this service can also be RACF controlled.

PKDS Record Create (CSNDKRC)

This callable service writes a new record to the PKDS.

Format

```
CALL CSNDKRC(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            label,
            token_length,
            token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. This parameter is ignored by ICSF.

rule_array

Direction: Input Type: String

This parameter is ignored by ICSF.

label

Direction: Input Type: String

The label of the record to be created. A 64 byte character string.

token_length

Direction: Input Type: Integer

The length of the field containing the token to be written to the PKDS. If zero is specified, a null token will be added to the PKDS. The maximum value of *token_length* is the maximum length of a private RSA or DSS token.

token

Direction: Input Type: String

Data to be written to the PKDS if *token_length* is non-zero. A RSA or DSS private token in either external or internal format, or a DSS or RSA public token.

Restriction

Caller must be task mode and must not be SRB mode.

Usage Note

PKA callable services must be enabled for you to use this service.

PKDS Record Delete (CSNDKRD)

Use PKDS record delete to delete a record from the PKDS.

Format

```
CALL CSNDKRD(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             label)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. This parameter is ignored by ICSF, except that its value must be 0, or 1.

rule_array

Direction: Input	Type: String

Keywords that provide control information to the callable service. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage.

Table 80. Keywords for PKDS Record Delete

Keyword	Meaning
Deletion Mode (optional) specifies whether the record is to be deleted entirely or whether only its contents are to be erased.	
LABEL-DL	Specifies that the record will be deleted from the PKDS entirely. This is the default deletion mode.
TOKEN-DL	Specifies that the only the contents of the record are to be deleted. The record will still exist in the PKDS, but will contain only binary zeroes.

label

Direction: Input	Type: String

The label of the record to be deleted. A 64 byte character string.

Restrictions

- Caller must be task mode and must not be SRB mode.
- This service cannot delete the PKDS record for a retained key.

Usage Note

PKA callable services must be enabled for you to use this service.

PKDS Record Read (CSNDKRR)

Reads a record from the PKDS and returns the content of the record. This is true even when the record contains a null PKA token.

Format

```
CALL CSNDKRR(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            label,
            token_length,
            token)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. This parameter is ignored by ICSF.

rule_array

Direction: Input	Type: String

This parameter is ignored by ICSF.

label

Direction: Input	Type: String

The label of the record to be read. A 64 byte character string.

token_length

Direction: Input/Output	Type: Integer

The length of the area to which the record is to be returned. On successful completion of this service, token_length will contain the actual length of the record returned.

token

Direction: Output	Type: String

Area into which the returned record will be written. The area should be at least as long as the record.

Restriction

Caller must be task mode and must not be SRB mode.

Usage Note

PKA callable services must be enabled for you to use this service.

PKDS Record Write (CSNDKRW)

Writes over an existing record in the PKDS.

Format

```
CALL CSNDKRW(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            label,
            token_length,
            token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you are supplying in the *rule_array* parameter. Its value must be 0 or 1.

rule_array

Direction: Input	Type: String

Keywords that provide control information to the callable service. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage.

Table 81. Keywords for PKDS Record Write

Keyword	Meaning
Write Mode (optional) specifies the circumstances under which the record is to be written.	
CHECK	Specifies that the record will be written only if a record of type NULL with the same label exists in the PKDS. If such a record exists, ICSF overwrites it. This is the default condition.
OVERLAY	Specifies that the record will be overwritten regardless of the current content of the record. If a record with the same label exists in the PKDS, ICSF overwrites it.

label

Direction: Input Type: String

The label of the record to be overwritten. A 64 byte character string.

token_length

Direction: Input Type: Integer

The length of the field containing the token to be written to the PKDS.

token

Direction: Input Type: String

The data to be written to the PKDS, which is a DSS or RSA private token in either external or internal format, or a DSS or RSA public token.

Restrictions

- Caller must be task mode and must not be SRB mode.
- This service cannot update a PKDS record for a retained key.

Usage Note

PKA callable services must be enabled for you to use this service.

Retained Key Delete (CSNDRKD)

Use the retained key delete callable service to delete a key that has been retained within the PCI Cryptographic Coprocessor. This service also deletes the record that contains the associated key token from the PKDS. It also allows the deletion of a retained key in the PCI Cryptographic Coprocessor even if there isn't a PKDS record, or deletion of a PKDS record for a retained key even if the PCI Cryptographic Coprocessor holding the retained key is not online. Use the *rule_array* parameter specifying the FORCE keyword and serial number of the PCI Cryptographic Coprocessor that contains the retained key to be deleted. If a PKDS record exists for the same label, but the serial number doesn't match the serial number in rule_array, the service will fail. If any applications still need the public key, use public key extract to create a public key token before deletion of the retained key.

Format

```
CALL CSNDRKD(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_label)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords supplied in the *rule_array* parameter. The value may be 0 or 2.

rule_array

Direction: Input	Type: Character String

This parameter may be FORCE and the PCI Cryptographic Coprocessor serial number.

key_label

Direction: Input	Type: String

A 64-byte label of a key that has been retained in a PCI Cryptographic Coprocessor.

Restriction

Caller must be task mode and must not be SRB mode.

Usage Notes

- ICSF calls the Security Server (RACF) to check authorization to use the Retained Key Delete service and the label of the key specified in key_label.
- Retained private keys are domain-specific. Only the LPAR domain that created a Retained private key can delete the key via the Retained Key Delete service.
- When a Retained key is deleted using the Retained Key Delete service, ICSF records this event in a type 82 SMF record with a subtype of 15.
- If the Retained key does not exist in the PCI Cryptographic Coprocessor and the PKDS record exists and the domain that created the retained key matches the domain of the requestor, ICSF deletes the PKDS record. This situation may occur if the PCI Cryptographic Coprocessor has been zeroized through TKE or the service processor.
- If a PKDS record containing the retained key exists but the PCI Cryptographic Coprocessor holding the retained key is not online, ICSF deletes the PKDS record if the FORCE keyword is specified.
- If the retained key exists on the specified PCI Cryptographic Coprocessor but there is no corresponding PKDS record, ICSF deletes the retained key from the PCI Cryptographic Coprocessor if the FORCE keyword is specified.

Retained Key List (CSNDRKL)

Use the retained key list callable service to list the key labels of those keys that have been retained within all currently active PCI Cryptographic Coprocessors.

Format

```
CALL CSNDRKL(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            key_label_mask
            retained_keys_count
            key_labels_count
            key_labels)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input   Type: Integer

The number of keywords supplied in the *rule_array* parameter. The value must be 0.

rule_array

Direction: Input   Type: Character String

This parameter is ignored by ICSF.

key_label_mask

Direction: Input   Type: String

A 64-byte key label mask that is used to filter the list of key names returned by the verb. You can use a wild card (*) to identify multiple keys retained within the PCI Cryptographic Coprocessor.

Note: If an asterisk (*) is used, it must be the last character in key_label_mask. There can only be one *.

retained_keys_count

Direction: Output   Type: Integer

An integer variable to receive the number of retained keys stored within all active PCI Cryptographic Coprocessors.

key_labels_count

Direction: Input/Output	Type: Integer

On input this variable defines the maximum number of key labels to be returned. On output this variable defines the total number of key labels returned. The value returned in the *retained_keys_count* variable can be larger if you have not provided for the return of a sufficiently large number of key labels in the *key_labels_count* field.

key_labels

Direction: Output	Type: String

A string variable where the key label information will be returned. This field must be at least 64 times the key label count value. The key label information is a string of zero or more 64-byte entries. The first 64-byte entry contains a PCI Cryptographic Coprocessor card serial number, and is followed by one or more 64-byte entries that each contain a key label of a key retained within that PCI Cryptographic Coprocessor. The format of the first 64-byte entry is as follows:

```
 /nnnnnnnnbbbbb...bbb
where
 "/" is the character "/" (EBCDIC: X'61')
 "nnnnnnnn" is the 8-byte PCI Cryptographic Coprocessor card
   serial number
 "bbbbb...bbb" is 55 bytes of blank pad characters
   (EBCDIC: X'40')
```

This information (64-byte card serial number entry followed by one or more 64-byte label entries) is repeated for each active PCI Cryptographic Coprocessor that contains retained keys that match the *key_label_mask*. All data returned is EBCDIC characters. The number of bytes of information returned is governed by the value specified in the *key_labels_count* field. The *key_labels* field must be large enough to hold the number of 64-byte labels specified in the *key_labels_count* field plus one 64-byte entry for each active PCI Cryptographic Coprocessor (a maximum of 64 PCI Cryptographic Coprocessors).

Restriction

Caller must be task mode and must not be SRB mode.

Usage Notes

- Not all CCA platforms may support multiple PCI Cryptographic Coprocessor cards. In the case where only one card is supported, the *key_labels* field will contain one or more 64-byte entries that each contain a key label of a key retained within the PCI Cryptographic Coprocessor. There will be no 64-byte entry or entries containing a PCI Cryptographic Coprocessor card serial number.
- ICSF calls RACF to check authorization to use the Retained Key List service.
- ICSF caller must be authorized to the *key_label_mask* name including the *.
- Retained private keys are domain-specific. ICSF lists only those keys that were created by the LPAR domain that issues the Retained Key List request.

Chapter 10. Utilities

This chapter presents utilities to perform the following tasks:

- Build DES key tokens
- Encipher plaintext using the Cipher Block Chaining (CBC) method
- Convert a binary string to a character string or a character string to a binary string
- Edit an ASCII string according to the editing rules of ANSI X9.9-4

This chapter describes the following callable services:

- “Character/Nibble Conversion (CSNBXBC and CSNBXCB)”
- “Code Conversion (CSNBXEA and CSNBXAE)” on page 307
- “X9.9 Data Editing (CSNB9ED)” on page 308

Character/Nibble Conversion (CSNBXBC and CSNBXCB)

Use these utilities to convert a binary string to a character string (CSNBXBC) or convert a character string to a binary string (CSNBXCB).

Format

```
CALL CSNBXBC(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             text_length,
             source_text,
             target_text,
             code_table)
```

```
CALL CSNBXCB(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             text_length,
             source_text,
             target_text,
             code_table)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Ignored	Type: Integer

Reserved field.

exit_data

Direction: Ignored	Type: String

Reserved field.

text_length

Direction: Input/Output	Type: Integer

On input, the *text_length* contains an integer that is the length of the *source_text*. The length must be a positive nonzero value. On output, *text_length* is updated with an integer that is the length of the *target_text*.

source_text

Direction: Input	Type: String

This parameter contains the string to convert.

target_text

Direction: Output	Type: String

The converted text that the callable service returns.

code_table

Direction: Input	Type: String

A 16-byte conversion table. The code table for binary to EBCDIC conversion is X'F0F1F2F3F4F5F6F7F8F9C1C2C3C4C5C6'.

Usage Notes

These services are structured differently from the other services. They run in the caller's address space in the caller's key and mode.

ICSF need not be active for you to run either of these services. No pre- or post-processing exits are enabled for these services, and no calls to RACF are issued when you run these services.

Code Conversion (CSNBXEA and CSNBXAE)

Use these utilities to convert ASCII data to EBCDIC data (CSNBXAE) or EBCDIC data to ASCII data (CSNBXEA).

Format

```
CALL CSNBXAE(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            text_length,
            source_text,
            target_text,
            code_table)
```

```
CALL CSNBXEA(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            text_length,
            source_text,
            target_text,
            code_table)
```

Parameters

return_code

Direction: Output
Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output
Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Ignored
Type: Integer

Reserved field.

exit_data

Direction: Ignored
Type: String

Reserved field.

text_length

Direction: Input	Type: Integer

The *text_length* contains an integer that is the length of the *source_text*. The length must be a positive nonzero value.

source_text

Direction: Input	Type: String

This parameter contains the string to convert.

target_text

Direction: Output	Type: String

The converted text that the callable service returns.

code_table

Direction: Input	Type: String

A 256-byte conversion table. When value is zero, this service uses the default code table. See Appendix G, "EBCDIC and ASCII Default Conversion Tables" on page 445 for contents of the default table.

Note: The Transaction Security System code table has 2 additional 8-byte fields that are not used in the conversion process. ICSF accepts either a 256-byte or a 272-byte code table, but uses only the first 256 bytes in the conversion.

Usage Notes

These services are structured differently than the other services. They run in the caller's address space in the caller's key and mode. ICSF need not be active for you to run either of these services. No pre- or post-processing exits are enabled for these services, and no calls to RACF are issued when you run these services.

X9.9 Data Editing (CSNB9ED)

Use this utility to edit an ASCII text string according to the editing rules of ANSI X9.9-4. It edits the text that the *source_text* parameter supplies according to the following rules. The rules are listed here in the order in which they are applied. It returns the result in the *target_text* parameter.

1. This service replaces each carriage-return (CR) character and each line-feed (LF) character with a single-space character.
2. It replaces each lowercase alphabetic character (a through z) with its equivalent uppercase character (A through Z).
3. It deletes all characters other than the following:
   - Alphabetics A...Z
   - Numerics 0...9
   - Space
   - Comma ,
   - Period .
   - Dash -

- Solidus /
- Asterisk *
- Open parenthesis (
- Close parenthesis)

4. It deletes all leading space characters.
5. It replaces all sequences of two or more space characters with a single-space character.

Format

```
CALL CSNB9ED(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             text_length,
             source_text,
             target_text)
```

Parameters

return_code

Direction: Output Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Ignored Type: Integer

Reserved field.

exit_data

Direction: Ignored Type: String

Reserved field.

text_length

Direction: Input/Output Type: Integer

On input, the *text_length* contains an integer that is the length of the *source_text*. The length must be a positive, nonzero value. On output, *text_length* is updated with an integer that is the length of the edited text.

source_text

Direction: Input	Type: String

This parameter contains the string to edit.

target_text

Direction: Output	Type: String

The edited text that the callable service returns.

Usage Notes

This service is structured differently from the other services. It runs in the caller's address space in the caller's key and mode.

ICSF need not be active for the service to run. There are no pre-processing or post-processing exits that are enabled for this service. While running, this service does not issue any calls to RACF.

Chapter 11. Trusted Key Entry Workstation Interfaces

For S/390 Enterprise Servers, the S/390 Multiprise, and the IBM @server zSeries, you can order an optional feature, the Trusted Key Entry (TKE) workstation. You can use this to load DES and PKA master keys, SYM-MK and ASYM-MK master keys on the PCI Cryptographic Coprocessor, and securely add operational key-encrypting keys and PIN keys to the CKDS. TKE uses the PKSC interface callable service (CSFPKSC) for support of the Cryptographic Coprocessor Feature and the PCI interface callable service (CSFPCI) for the support of the PCI Cryptographic Coprocessor.

This chapter describes the following callable services:

- "PCI Interface Callable Service (CSFPCI)"
- "PKSC Interface Callable Service (CSFPKSC)" on page 315

PCI Interface Callable Service (CSFPCI)

TKE uses this callable service to send a request to a specific PCI card queue and remove the corresponding response when complete. This service also allows the TKE workstation to query the list of access control points which may be enabled or disabled by a TKE user. This service is synchronous. The return and reason codes reflect the success or failure of the NQAP and DQAP functions rather than the success or failure of the actual PCI request.

Format

```
CALL CSFPCI(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            target_pci_coprocessor,
            target_pci_coprocessor_serial_number,
            request_block_length,
            request_block,
            request_data_block_length,
            request_data_block,
            reply_block_length,
            reply_block,
            reply_data_block_length,
            reply_data_block,
            masks_length,
            masks_data)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. See Appendix A, "ICSF and TSS Return and Reason Codes", for a list of return codes.

reason_code

Direction: Output Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. See Appendix A, "ICSF and TSS Return and Reason Codes" for a list of reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you are supplying in *rule_array*. The value must be 1.

rule_array

Direction: Input Type: String

Keyword that provides control information to callable services. The keyword is left-justified in an 8-byte field and padded on the right with blanks. The keyword must be in contiguous storage. The keywords listed below are mutually exclusive.

Table 82. Keywords for PCI Interface Callable Service

Keyword	Meaning
ACPOINTS	Queries the list of access control points which may be enabled or disabled by a TKE user.
ACTIVECP	This keyword is a request to call the PCI card initialization code to revalidate the PCI cards. After the PCI card initialization is completed, both the 64-bit mask indicating which of the PCI cards are online and 64-bit mask indicating which of the PCI cards are active will be returned. This keyword is used by the TKE workstation code after the ACTIVATE portion of the domain zeroize command. This is to ensure that the status of the PCI card is accurately reflected to the users. See the *masks_data* parameter description for more information.
APNUM	Specifies the *target_pci_coprocessor* field to be used.
SERIALNO	Specifies the *target_pci_coprocessor_number* field to be used

Table 82. Keywords for PCI Interface Callable Service (continued)

Keyword	Meaning
PCIMASKS	This keyword is a request to return both the 64-bit mask indicating which of the PCI cards are online and 64-bit mask indicating which of the PCI cards are active. See the *masks_data* parameter description for more information.

Note: When the PCIMASKS and ACTIVEP keywords are specified, the *request_data_block_length*, *request_data_block*, *reply_data_block_length*, and the *reply_data_block* parameters are ignored.

target_pci_coprocessor

Direction: Input Type: Integer

The PCI Cryptographic Coprocessor card to which this request is directed. Value is 1 - 64.

target_pci_coprocessor_serial_number

Direction: Input Type: String

The PCI Cryptographic Coprocessor card serial number to which the request is directed. This parameter may be used instead of the *target_pci_coprocessor*. The length is 8 bytes.

request_block_length

Direction: Input/Output Type: Integer

Length of CPRB and the request block in the *request_block* field. The maximum length allowed is 5,500 bytes.

request_block

Direction: Input Type: String

PCI Cryptographic Coprocessor command or query request for the target PCI Cryptographic Coprocessor. This is the complete CPRB and request block to be processed by the PCI Cryptographic Coprocessor.

request_data_block_length

Direction: Input Type: Integer

Length of request data block in the *request_data_block* field. The maximum length allowed is 6,400 bytes. The length field must be a multiple of 4.

request_data_block

Direction: Input Type: String

The data that accompanies the *request_block* field.

reply_block_length

Direction: Input/Output	Type: Integer

Length of CPRB and the reply block in the *reply_block* field. The maximum length allowed is 5,500 bytes. This field is updated on output with the actual length of the *reply_block* field.

reply_block

Direction: Output	Type: String

PCI Cryptographic Coprocessor reply from the target PCI Cryptographic Coprocessor. This is the CPRB and reply block that has been processed by the PCI Cryptographic Coprocessor.

reply_data_block_length

Direction: Input/Output	Type: Integer

Length of reply block in the *reply_data_block* field. The maximum length allowed is 6,400 bytes. This field is updated on output with the actual length of the *reply_data_block* field. This length field must be a multiple of 4. For the ACPOINTS keyword, the minimum length is 2572 bytes.

reply_data_block

Direction: Output	Type: String

The data that accompanies the *reply_block* field.

masks_length

Direction: Input	Type: Integer

Length of the reply data being returned in the *masks_data* field. The length must be 32 bytes. This field is only valid when the input *rule_array* keyword is PCIMASKS or ACTIVECP. For all other *rule_array* keywords, this field is ignored.

masks_data

Direction: Output	Type: String

The data being returned for all requests. The first 8 bytes indicate the count of the PCI cards online. The second 8 bytes indicate a bit mask of the actual PCI cards brought online. The third 8 bytes indicate the count of the PCI cards active. The fourth 8 bytes indicate a bit mask of the actual PCI cards that are active. For the ACTIVECP keyword, if the PCI card initialization failed, the appropriate return code and reason code is issued and the *masks_data* field will contain zeros.

Restriction

The caller must be in task mode, not in SRB mode.

Usage Note

The *target_pci_coprocessor*, the *target_pci_coprocessor_serial_number*, the *request_block*, the *reply_block*, the *request_block_data_block*, and the *reply_block_data_block*, are recorded in SMF Record Type 82, subtype 16.

PKSC Interface Callable Service (CSFPKSC)

TKE uses this callable service to send a request to a specific cryptographic module and receive a corresponding response when processing is complete. The service is synchronous. Note that the return and reason codes reflect the success or failure of CSFPKSC's interaction with the cryptographic module rather than the success or failure of the cryptographic module request. The response block contains the results of the cryptographic module request.

Format

```
CALL CSFPKSC(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            target_crypto_module,
            request_length,
            request,
            response)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, "ICSF and TSS Return and Reason Codes" lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, "ICSF and TSS Return and Reason Codes" lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

target_crypto_module

Direction: Input	Type: Integer

Cryptographic module to which this request is directed. Value is 0 or 1.

request_length

Direction: Input	Type: Integer

Length of request message in the *request* field. The maximum length allowed is 1024 bytes.

request

Direction: Input	Type: String

PKSC command or query request for the target cryptographic module. This is the complete architected command or query for the cryptographic module to process.

response

Direction: Output	Type: String

Area where the PKSC response from the target cryptographic module is returned to the caller. The area returned can be up to 512 bytes.

Restrictions

The caller must be in task mode, not in SRB mode.

The format and content of the PKSC request and response areas are proprietary IBM hardware information that may be licensed. Customers interested in this information may contact the IBM Director of Licensing. For the address, refer to “Notices” on page 453.

Chapter 12. Managing Keys According to the ANSI X9.17 Standard

This chapter describes the callable services that support the ANSI X9.17 key management standard:

- “ANSI X9.17 EDC Generate (CSNAEGN)”
- “ANSI X9.17 Key Export (CSNAKEX)” on page 319
- “ANSI X9.17 Key Import (CSNAKIM)” on page 324
- “ANSI X9.17 Key Translate (CSNAKTR)” on page 328
- “ANSI X9.17 Transport Key Partial Notarize (CSNATKN)” on page 333

The following callable services, that are described in other sections of this book, also support the ANSI X9.17 key management standard:

- “Key Generate (CSNBKGN)” on page 82
- “Key Part Import (CSNBKPI)” on page 97
- “Key Token Build (CSNBKTB)” on page 109

ANSI X9.17 EDC Generate (CSNAEGN)

Use the ANSI X9.17 EDC generate callable service to generate an error detection code (EDC) on a text string. The service calculates the EDC by by using a key value of X'0123456789ABCDEF' to generate a MAC on the specified text string, as defined by the ANSI X9.17 standard.

Format

```
CALL CSNAEGN(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             text_length,
             text,
             chaining_vector,
             EDC)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value must be 0.

rule_array

Direction: Input Type: String

Keywords that provide control information to the callable service. Currently there are no keywords that are defined for this variable, but you must declare the variable. To do so, declare an area of blanks of any length.

text_length

Direction: Input Type: Integer

The length of the user-supplied *text* parameter for which the service should calculate the EDC.

text

Direction: Input Type: String

The application-supplied text field for which the service is to generate the EDC.

chaining_vector

Direction: Input/Output Type: String

An 18-byte string that ICSF uses as a system work area. The chaining vector permits data to be chained from one call to another. ICSF ignores the information in this field, but you must declare an 18-byte string.

EDC

Direction: Output Type: String

A 9-byte field where the callable service returns the EDC generated as two groups of four ASCII-encoded hexadecimal characters that are separated by an ASCII space character.

Usage Notes

The ANSI X9.17 standard states that for EDC, before the service generates the MAC the caller must first edit the input text according to section 4.3 of ANSI X9.9-1982. It is the caller's responsibility to do the editing before calling the ANSI X9.17 EDC generate service. If the supplied text is not a multiple of 8, the service pads the text with X'00' up to a multiple of 8, as specified in ANSI X9.9-1.

To use this service you must have the ANSI system keys installed in the CKDS.

ANSI X9.17 Key Export (CSNAKEX)

Use the ANSI X9.17 key export callable service to export a DATA key or a pair of DATA keys, along with an ANSI key-encrypting key (AKEK), using the ANSI X9.17 protocol. This service converts a single DATA key, or combines two DATA keys, into a single MAC key. You can use the MAC key in either, or both, the MAC generation, or MAC verification service to authenticate the service message. In addition, this service also supports the export of a CCA IMPORTER or EXPORTER KEK.

If you export only DATA keys, the DATA keys are exported encrypted under the specified transport AKEK. You have the option of applying the ANSI X9.17 key offset or key notarization process to the transport AKEK.

If you export both DATA keys and an AKEK, the DATA keys are exported encrypted under the key-encrypting key that is also being exported. The AKEK is exported encrypted under the specified transport AKEK. You have the option of applying the ANSI X9.17 key offset or key notarization process to the transport AKEK. The ANSI X9.17 key offset process is applied to the source AKEK. Use the CKT keyword to specify whether to use an offset of 0 or 1. Use an offset of 0 when sending the DATA key to a key translation center along with a transport AKEK.

Note: You must create the cryptographic service message and maintain the offset counter value that is associated with the AKEK.

Format

```
CALL CSNAKEX(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            origin_identifier,
            destination_identifier,
            source_data_key_1_identifier,
            source_data_key_2_identifier,
            source_key_encrypting_key_identifier,
            transport_key_identifier,
            outbound_KEK_count,
            target_data_key_1,
            target_data_key_2,
            target_key_encrypting_key,
            MAC_key_token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicates specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value can be 0 to 4. If you specify 0, the callable service does not perform either notarization or offset.

rule_array

Direction: Input Type: String

Zero to four keywords that provide control information to the callable service. See the list of keywords in Table 83. The keywords must be in 8 to 32 bytes of contiguous storage. Left-justify each keyword in its own 8-byte location and pad on the right with blanks. You must specify this parameter even if you specify no keyword.

Table 83. Keywords for ANSI X9.17 Key Export Rule Array

Keyword	Meaning
Notarization and Offset Rule (optional with no defaults)	
CPLT-NOT	Complete ANSI X9.17 notarization using the value obtained from the *outbound_KEK_count* parameter. The transport key that the *transport_key_identifier* specifies must be partially notarized.
NOTARIZE	Perform notarization processing using the values obtained from the *origin_identifier*, *destination_identifier*, and *outbound_KEK_count* parameters.
OFFSET	Perform ANSI X9.17 key offset processing using the origin counter value obtained from the *outbound_KEK_count* parameter.
Parity Rule (optional)	
ENFORCE	Stop processing if any source keys do not have odd parity. This is the default value.
IGNORE	Ignore the parity of the source key.
Source Key Rule (optional)	
CCA-EXP	Export a CCA EXPORTER KEK. Requires NOCV keys to be enabled.
CCA-IMP	Export a CCA IMPORTER KEK. Requires NOCV keys to be enabled.
1-KD	Export one DATA key. This is the default parameter.
1-KD+KK	Export one DATA key and a single-length AKEK.
1-KD+*KK	Export one DATA key and a double-length AKEK.
2-KD	Export two DATA keys.
2-KD+KK	Export two DATA keys and a single-length AKEK.
2-KD+*KK	Export two DATA keys and a double-length AKEK.
Data Key Offset Value (optional)	
CKT	Valid only when a key-encrypting key is being exported along with a DATA key. If this keyword is specified, any DATA keys being exported are encrypted under the key-encrypting key using an offset value of 0. If this keyword is not specified (this is the default), any DATA keys being exported are encrypted under the key-encrypting key using an offset value of 1. The CKT keyword is not valid with CCA-IMP or CCA-EXP keywords.

origin_identifier

Direction: Input Type: String

This parameter is valid if the NOTARIZE keyword is specified. It specifies an area that contains a 16-byte string that contains the origin identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. This parameter must be a minimum of four, non-space characters. ICSF ignores this parameter if you specify the OFFSET or CPLT-NOT keyword in the *rule_array* parameter.

destination_identifier

Direction: Input Type: String

This parameter is valid if the NOTARIZE keyword is specified. It specifies an area that contains a 16-byte string. The 16-byte string contains the destination identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. This parameter must be a minimum of four, non-space characters. ICSF ignores this parameter if you specify the OFFSET or CPLT-NOT keyword in the *rule_array* parameter.

source_data_key_1_identifier

Direction: Input/Output Type: String

A 64-byte area that contains an internal token, or the label of a CKDS entry that contains a DATA key. ICSF ignores this field if you specify CCA-EXP or CCA-IMP in the *rule_array* parameter.

source_data_key_2_identifier

Direction: Input/Output Type: String

A 64-byte area that contains an internal token, or the label of a CKDS entry that contains a DATA key. This parameter is valid only if you specify 2-KD, 2-KD+KK, or 2-KD+*KK as the source key rule keyword on the *rule_array* parameter. ICSF ignores this parameter if you specify other source key rule keywords, or if you specify CCA-EXP or CCA-IMP in the *rule_array* parameter.

source_key_encrypting_key_identifier

Direction: Input/Output Type: String

A 64-byte area that contains an internal token, or the label of a CKDS entry that contains either an AKEK, a CCA IMPORTER, or a CCA EXPORTER key. If this parameter contains an AKEK, you must specify 1-KD+KK, 2-KD+KK, 1-KD+*KK, or 2-KD+*KK for the source key rule on the *rule_array* parameter. If this parameter contains a CCA IMPORTER or CCA EXPORTER key, you must specify CCA-IMP or CCA-EXP, respectively, for the source key rule on the *rule_array* parameter. ICSF ignores this field if you specify any other source key rule keywords.

transport_key_identifier

Direction: Input/Output Type: String

A 64-byte area that contains either an internal token or a label that refers to an internal token for an AKEK.

outbound_KEK_count

Direction: Input Type: String

An 8-byte area that contains an ASCII count that is used in the notarization process. The count is an ASCII character string, left-justified, and padded on the right by ASCII space characters. ICSF interprets a single ASCII space character as a zero counter. The maximum value is 99999999.

target_data_key_1

Direction: Output Type: String

A 16-byte area where the exported data key 1 is returned. The enciphered key is an ASCII-encoded hexadecimal string.

target_data_key_2

Direction: Output Type: String

A 16-byte area where the exported data key 2 is returned. The enciphered key is an ASCII-encoded hexadecimal string. This key is returned if 2-KD, 2-KD+KK, or 2-KD+*KK is specified in the *rule_array* parameter.

target_key_encrypting_key

Direction: Output Type: String

If the *rule_array* parameter specifies 1-KD+KK, 2-KD+KK, 1-KD+*KK, or 2-KD+*KK, this parameter specifies a 32-byte area that contains the exported AKEK. If the *rule_array* parameter specifies CCA-IMP or CCA-EXP, this parameter specifies a 32-byte area that contains the exported key-encrypting key (KEK). The enciphered key is an ASCII-encoded hexadecimal string. If the *rule_array* parameter specifies 1-KD+KK or 2-KD+KK, the 16-byte ASCII-encoded output is left-justified in the field and the rest of the field remains unchanged.

MAC_key_token

Direction: Output Type: String

A 64-byte area that contains an internal token for a MAC key that is intended for use in the MAC generation or MAC verification process. This field is the EXCLUSIVE OR of the two supplied DATA keys when the source key rule in the *rule_array* parameter specifies 2-KD, 2-KD+KK, or 2-KD+*KK. When the source key rule specifies 1-KD, the DATA key is converted to a MAC key and returned as an internal token in this field.

Usage Note

You must install the ANSI system keys in the CKDS to use this service.

ANSI X9.17 Key Import (CSNAKIM)

Use the ANSI X9.17 key import callable service to import a DATA key or a pair of DATA keys, along with an ANSI key-encrypting key (AKEK), using the ANSI X9.17 protocol. This service converts a single DATA key, or combines two DATA keys, into a single MAC key. The MAC key can be used in either, or both, the MAC generation or the MAC verification service to authenticate the service message. In addition, this service also supports the import of the KEK to a CCA IMPORTER or EXPORTER KEK, as well as an AKEK.

If you are importing only DATA keys, this service assumes that the DATA keys are encrypted under the specified transport AKEK. You have the option of applying the ANSI X9.17 key offset or key notarization process to the transport AKEK.

If you are importing both DATA keys and an AKEK, this service assumes that the AKEK is encrypted under the specified transport AKEK. This service also assumes that the DATA keys are encrypted under the source AKEK that is also being imported. You have the option of applying the ANSI X9.17 key offset or key notarization process to the transport AKEK. ICSF applies the ANSI X9.17 key offset process to the source AKEK with an offset of 1.

Note: You must create the cryptographic service message and maintain the offset counter value that is associated with the AKEK.

Format

```
CALL CSNAKIM(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            origin_identifier,
            destination_identifier,
            source_data_key_1,
            source_data_key_2,
            source_key_encrypting_key,
            inbound_KEK_count,
            transport_key_identifier,
            target_data_key_1,
            target_data_key_2,
            target_key_encrypting_key,
            MAC_key_token)
```

Parameters

return_code

Direction: Output   Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output   Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output   Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output   Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input   Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value can be 0 to 3. If you specify 0, ICSF does not perform either notarization or offset.

rule_array

Direction: Input   Type: String

Zero to three keywords that provide control information to the callable service. See the list of keywords in Table 84. The keywords must be in 8 to 24 bytes of contiguous storage. Each of the keywords must be left-justified in its own 8-byte location and padded on the right with blanks. You must specify this parameter even is you do not specify a keyword.

Table 84. Keywords for ANSI X9.17 Key Import Rule Array

Keyword	Meaning
Notarization and Offset Rule (optional with no defaults)	
CPLT-NOT	Complete ANSI X9.17 notarization using the value obtained from the *inbound_KEK_count* parameter. The transport key that the *transport_key_identifier* specifies must be partially notarized.
NOTARIZE	Perform notarization processing using the values obtained from the *origin_identifier*, *destination_identifier*, and *inbound_KEK_count* parameters.
OFFSET	Perform ANSI X9.17 key offset processing using the origin counter value obtained from the *inbound_KEK_count* parameter.

Table 84. Keywords for ANSI X9.17 Key Import Rule Array (continued)

Keyword	Meaning
Parity Rule (optional)	
ENFORCE	Stop processing if any source keys do not have odd parity. This is the default value.
IGNORE	Ignore the parity of the source key.
Source Key Rule (optional)	
CCA-EXP	Import a key-encrypting key as a CCA EXPORTER. Requires NOCV keys to be enabled.
CCA-IMP	Import a key-encrypting key as a CCA IMPORTER. Requires NOCV keys to be enabled.
1-KD	Import one DATA key. This is the default parameter.
1-KD+KK	Import one DATA key and a single-length AKEK.
1-KD+*KK	Import one DATA key and a double-length AKEK.
2-KD	Import two DATA keys.
2-KD+KK	Import two DATA keys and a single-length AKEK.
2-KD+*KK	Import two DATA keys and a double-length AKEK.

origin_identifier

Direction: Input   Type: String

This parameter is valid if you specify the NOTARIZE keyword in the *rule_array* parameter. It specifies an area that contains a 16-byte string that contains the origin identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. The string must be a minimum of four, non-space characters. This parameter is ignored if the OFFSET or CPLT-NOT keyword is specified.

destination_identifier

Direction: Input   Type: String

This parameter is valid if you specify the NOTARIZE keyword in the *rule_array* parameter. It specifies an area that contains a 16-byte string that contains the destination identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. It must be a minimum of four non-space characters. This parameter is ignored if the OFFSET or CPLT-NOT keyword is specified.

source_data_key_1

Direction: Input   Type: String

A 16-byte area that contains the enciphered DATA key to be imported. You must supply the DATA key as an ASCII-encoded hexadecimal string. The field is ignored if the *rule_array* parameter specifies CCA-IMP or CCA-EXP.

source_data_key_2

Direction: Input   Type: String

A 16-byte area that contains the second enciphered DATA key to be imported. This parameter is valid only if the *rule_array* parameter specifies KK, or 2-KD+*KK. You must supply the key as an ASCII-encoded hexadecimal string. This field is ignored if the *rule_array* parameter specifies other source key rules.

source_key_encrypting_key

Direction: Input Type: String

A 16- or 32-byte area that contains an enciphered AKEK, if the *rule_array* parameter specifies either 1-KD+KK, 2-KD+KK, 1-KD+*KK, or 2-KD+*KK. This parameter specifies a KEK, if the *rule_array* parameter specifies either CCA-IMP or CCA-EXP. The area is 16 bytes if the *rule_array* parameter specifies a single-length AKEK (1-KD+KK or 2-KD+KK). The area is 32 bytes if the *rule_array* parameter specifies a double-length AKEK (1-KD+*KK or 2-KD+*KK). You must supply the key as an ASCII-encoded hexadecimal string. This field is ignored if the *rule_array* parameter specifies 1-KD or 2-KD.

inbound_KEK_count

Direction: Input Type: String

An 8-byte area that contains an ASCII count for use in the notarization process. The count is an ASCII character string, left-justified, and padded on the right by space characters. ICSF interprets a single space character as a zero counter. The maximum value is 99999999.

transport_key_identifier

Direction: Input/Output Type: String

A 64-byte area that contains an internal token or a label that refers to an internal token for an AKEK.

target_data_key_1

Direction: Output Type: String

A 64-byte area where the imported data key 1 is returned as an ICSF internal key token. ICSF does not support the direct import by label.

target_data_key_2

Direction: Output Type: String

A 64-byte area where the imported data key 2 is returned as an ICSF internal key token. ICSF does not support the direct import by label. This key is returned if 2-KD, 2-KD+KK, or 2-KD+*KK is specified in the *rule_array* parameter.

target_key_encrypting_key

Direction: Output Type: String

A 64-byte area where the imported key-encrypting key is returned as an ICSF internal key token. If the *rule_array* parameter specifies 1-KD+KK, 1-KD+*KK, 2-KD+KK, or 2-KD+*KK, the internal key token contains an AKEK. If the *rule_array* parameter specifies either CCA-IMP or CCA-EXP, the internal token contains a CCA IMPORTER or a CCA EXPORTER, respectively.

MAC_key_token

Direction: Output	Type: String

A 64-byte area that contains an internal token for a MAC key that is intended for use in the MAC generation or MAC verification function. This field is the EXCLUSIVE OR of the two imported DATA keys if the source key rule in the *rule_array* parameter specifies 2-KD, 2-KD+KK, or 2-KD+*KK. If the source key rule in the *rule_array* parameter specifies 1-KD, ICSF converts the DATA key to a MAC key and returns it as an internal token in this field.

Usage Note

You must install the ANSI system keys in the CKDS to use this service.

ANSI X9.17 Key Translate (CSNAKTR)

Use the ANSI X9.17 key translate callable service to translate a key from encryption under one AKEK to encryption under another AKEK. In a single service call you can translate either one or two encrypted DATA keys, or a single encrypted key-encrypting key. In addition, this service also imports the supplied DATA keys. If the *rule_array* parameter specifies 2-KD, this service exclusive-ORs the two imported DATA keys and converts the result into a MAC key, which it returns in the *MAC_key_token* field. The MAC key is used to perform MAC processing on the service message. If the *rule_array* specifies keywords 1-KD and 2-KD, ICSF translates only DATA keys. The service uses the inbound transport key-encrypting key to decrypt the DATA keys, and uses the outbound transport key-encrypting key to reencrypt the DATA keys. The service uses the ANSI X9.17 key offset process during decryption or importing. The service can use the ANSI X9.17 notarization process during reencryption or exporting of the DATA keys.

If the *rule_array* parameter specifies 1-KD+KK or 1-KD+*KK , the service translates only the AKEK. The service uses the inbound transport key-encrypting key to decrypt or import the input AKEK, applying the ANSI X9.17 offset process. The service uses the outbound transport key-encrypting key to reencipher or export the AKEK, with or without applying the optional ANSI X9.17 notarization process. ICSF uses the inbound key-encrypting key that is being translated to import the supplied DATA key, applying the ANSI X9.17 offset processing only with an offset of 0. The DATA key is imported as above then converted to a MAC key token and returned in the *MAC_key_token* field.

Format

```
CALL CSNAKTR(
             return_code,
             reason_code,
             exit_data_length,
             exit_data,
             rule_array_count,
             rule_array,
             inbound_KEK_count,
             inbound_transport_key_identifier,
             inbound_data_key_1,
             inbound_data_key_2,
             inbound_key_encrypting_key,
             outbound_origin_identifier,
             outbound_destination_identifier,
             outbound_KEK_count,
             outbound_transport_key_identifier,
             outbound_data_key_1,
             outbound_data_key_2,
             outbound_key_encrypting_key,
             MAC_key_token)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. The value can be 0 to 3. If you specify 0, the service does not perform notarization or offset.

rule_array

Direction: Input Type: String

Zero to three keywords that provide control information to the callable service. See the list of keywords in Table 85. The keywords must be in 8 to 24 bytes of contiguous storage. Each of the keywords must be left-justified in its own 8-byte location and padded on the right with blanks. You must specify this parameter even if do not specify any keywords.

Table 85. Keywords for ANSI X9.17 Key Translate Rule Array

Keyword	Meaning
Notarization Rule (optional with no defaults)	
CPLT-NOT	Complete ANSI X9.17 notarization using the value obtained from the *outbound_KEK_count* parameter. The outbound transport key specified must be partially notarized.
NOTARIZE	Perform notarization processing using the values obtained from the *outbound_origin_identifier*, the *outbound_destination_identifier*,and the *outbound_KEK_count*.
Parity Rule (optional)	
ENFORCE	Stop processing if any source keys do not have odd parity. This is the default value.
IGNORE	Ignore the parity of the source key.
Source Key Rule (optional)	
1-KD	Import and translate one DATA key. This is the default parameter.
1-KD+KK	Import and translate one DATA key and a single-length AKEK.
1-KD+*KK	Import and translate one DATA key and a double-length AKEK.
2-KD	Import and translate two DATA keys.

inbound_KEK_count

Direction: Input Type: String

An 8-byte area that contains an ASCII count for use in the offset process. The count is an ASCII character string, left-justified, and padded on the right by space characters. ICSF interprets a single space character as a zero counter. The maximum value is 99999999.

inbound_transport_key_identifier

Direction: Input/Output Type: String

A 64-byte area that contains either an internal token, or a label that refers to an internal token for an AKEK.

inbound_data_key_1

Direction: Input Type: String

A 16-byte area that contains the enciphered DATA key that the service is importing and translating. You must specify the DATA key as an ASCII-encoded hexadecimal string.

inbound_data_key_2

Direction: Input Type: String

A 16-byte area that contains the second enciphered DATA key that the service is importing and translating. This field is valid if the *rule_array* parameter specifies 2-KD. You must supply the key as an ASCII-encoded hexadecimal string. This field is ignored if the *rule_array* parameter specifies other source key rules.

inbound_key_encrypting_key

Direction: Input Type: String

A 16- or 32-byte area that contains an enciphered AKEK that the service is to translate. The area is 16 bytes if the *rule_array* parameter specifies a source key rule of single-length AKEK. The area is 32 bytes if the source key rule specifies a double-length AKEK (1-KD+*KK). You must supply the key as an ASCII-encoded hexadecimal string. ICSF ignores this field if the *rule_array* specifies either 1-KD or 2-KD.

outbound_origin_identifier

Direction: Input Type: String

This parameter is valid if the *rule_array* parameter specifies a keyword of NOTARIZE. It specifies an area that contains a 16-byte string that contains the origin identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. The string must be a minimum of four non-space characters. ICSF ignores this field if the *rule_array* parameter specifies a keyword of CPLT-NOT.

outbound_destination_identifier

Direction: Input Type: String

This parameter is valid if the *rule_array* parameter specifies a keyword of NOTARIZE. It specifies an area that contains a 16-byte string that contains the destination identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. The string must be a minimum of four non-space characters. This parameter is ignored if the *rule_array* parameter specifies a keyword of CPLT-NOT.

outbound_KEK_count

Direction: Input Type: String

An 8-byte area that contains an ASCII count for use in the notarization process. The count is an ASCII character string, left-justified, and padded on the right by space characters. ICSF interprets a single space character as a zero counter. The maximum value is 99999999.

outbound_transport_key_identifier

Direction: Input/Output	Type: String

A 64-byte area that contains either an internal token, or a label that refers to an internal token for an AKEK.

outbound_data_key_1

Direction: Output	Type: String

A 16-byte area where the service returns the translated data key 1 an ASCII-encoded hexadecimal string. The service returns the key only if the *rule_array* specifies 1-KD or 2-KD. ICSF ignores this field if the *rule_array* parameter specifies either 1-KD+KK or 1-KD+*KK.

outbound_data_key_2

Direction: Output	Type: String

A 16-byte area where the service returns the translated data key 2 as an ASCII-encoded hexadecimal string. The service returns the key only if the *rule_array* parameter specifies 2-KD. ICSF ignores this field if the *rule_array* parameter specifies 1-KD, 1-KD+KK, or 1-KD+*KK.

outbound_key_encrypting_key

Direction: Output	Type: String

A 16- or 32-byte area that contains the enciphered, translated AKEK. The area is 16 bytes if the *rule_array* parameter specifies a single-length AKEK (1-KD+KK). The area is 32 bytes if the *rule_array* parameter specifies a double-length AKEK (1-KD+*KK). The service returns the key as an ASCII-encoded hexadecimal string. ICSF ignores this field if the *rule_array* parameter specifies either 1-KD or 2-KD.

MAC_key_token

Direction: Output	Type: String

A 64-byte area that contains an internal token for a MAC key that is intended for use in the MAC generation or MAC verification process. This field is the EXCLUSIVE OR of the two imported DATA keys when the *rule_array* parameter specifies 2-KD for the source key rule. If the *rule_array* parameter specifies 1-KD, the service returns the imported key in this field as an ICSF internal key token.

Usage Note

You must install the ANSI system keys in the CKDS to use this service.

ANSI X9.17 Transport Key Partial Notarize (CSNATKN)

Use the ANSI X9.17 transport key partial notarize callable service to preprocess an ANSI X9.17 transport key-encrypting key with origin and destination identifiers. ICSF completes the notarization process when you use the partially notarized key in the ANSI X9.17 key export, ANSI X9.17 key import, or ANSI X9.17 key translate services and specify the CPLT-NOT *rule_array* keyword.

Note: You cannot reverse the partial notarization process. If you want to keep the original value of the AKEK, you must record the value.

Format

```
CALL CSNATKN(
            return_code,
            reason_code,
            exit_data_length,
            exit_data,
            rule_array_count,
            rule_array,
            origin_identifier,
            destination_identifier,
            source_transport_key_identifier,
            target_transport_key_identifier)
```

Parameters

return_code

Direction: Output	Type: Integer

The return code specifies the general result of the callable service. Appendix A, “ICSF and TSS Return and Reason Codes” lists the return codes.

reason_code

Direction: Output	Type: Integer

The reason code specifies the result of the callable service that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “ICSF and TSS Return and Reason Codes” lists the reason codes.

exit_data_length

Direction: Input/Output	Type: Integer

The length of the data that is passed to the installation exit. The length can be from X'00000000' to X'7FFFFFFF' (2 gigabytes). The data is identified in the *exit_data* parameter.

exit_data

Direction: Input/Output	Type: String

The data that is passed to the installation exit.

rule_array_count

Direction: Input	Type: Integer

The number of keywords you supplied in the *rule_array* parameter. Currently no *rule_array* keywords are defined; thus, this field must be set to 0.

rule_array

Direction: Input	Type: String

Currently, no *rule_array* keywords are defined for this service. You must still specify this parameter for possible future use.

origin_identifier

Direction: Input	Type: String

A 16-byte string that contains the origin identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. The string must be a minimum of four non-space characters.

destination_identifier

Direction: Input	Type: String

A 16-byte string that contains the destination identifier that is defined in the ANSI X9.17 standard. The string must be ASCII characters, left-justified, and padded on the right by space characters. The string must be a minimum of four non-space characters.

source_transport_key_identifier

Direction: Input/Output	Type: String

A 64-byte area that contains either an internal token, or a label of an internal token for an AKEK that permits notarization.

target_transport_key_identifier

Direction: Output	Type: String

A 64-byte area where the internal token of a partially notarized AKEK will be returned. This AKEK cannot be used directly as a notarizing KEK until the notarization process has been completed. To do this, specify CPLT-NOT as the *rule_array* keyword in any service in which you intend to use this key as a notarizing KEK.

Usage Note

You must install the ANSI system keys in the CKDS to use this service.

Appendix A. ICSF and TSS Return and Reason Codes

This appendix includes the following information:

- Return codes and reason codes issued on the completion of a call to an ICSF callable service
- Return codes and reason codes issued on the completion of a process on a PCI Cryptographic Coprocessor
- Conversion tables showing the relationship between ICSF and Transaction Security System return and reason codes. ICSF or TSS return and reason codes can be specified in the installation options dataset on the REASONCODES parameter. If the REASONCODES option is not specified, the default of REASONCODES(ICSF) is used.

ICSF Return Codes and Reason Codes

This section describes the ICSF return codes and reason codes and also lists ICSF to TSS return codes and reason codes. Each ICSF return code returns unique reason codes to your application program. The reason codes associated with each return code are described in the following sections. The reason code tables present the ICSF hexadecimal code followed by the decimal code in parenthesis. If there is a 1-to-1 mapping, the codes will be converted. If there is not a map to TSS, the column will be blank. If there are multiple mappings, they will be listed as reference only and will not be converted.

Return Codes

Table 86 lists return codes from the ICSF callable services.

Table 86. ICSF Return Codes

Return Code Hex (Decimal)	Description
Return Code 0 (0)	The call to the service or PCI Cryptographic Coprocessor was successfully processed. See the reason code for more information.
Return Code 4 (4)	The call to the service or PCI Cryptographic Coprocessor was successfully processed, but some minor event occurred during processing. See the reason code for more information. **User action**: Review the reason code.
Return Code 8 (8)	The call to the service or PCI Cryptographic Coprocessor was unsuccessful. The parameters passed into the call are unchanged, except for the return code and reason code. There are rare examples where output areas are filled, but their contents are not guaranteed to be accurate. These are described under the appropriate reason code descriptions. The reason code identifies which error was found. **User action**: Review the reason code, correct the problem, and retry the call.
Return Code C (12)	The call to the service or PCI Cryptographic Coprocessor could not be processed because ICSF was not active, ICSF found something wrong in its environment, a TSS security product is not available, or a processing error occurred in a TSS product. The parameters passed into the call are unchanged, except for the return code and reason code. **User action**: Review the reason code and take the appropriate action.

Table 86. ICSF Return Codes (continued)

Return Code Hex (Decimal)	Description
Return Code 10 (16)	The call to the service or PCI Cryptographic Coprocessor could not be processed because ICSF found something seriously wrong in its environment or a processing error occurred in the PCI Cryptographic Coprocessor. The parameters passed into the call are unchanged, except for the return code and reason code. **User action**: Review the reason code and contact your system programmer.

ICSF Reason Codes for Return Code 0 (0)

Table 87 lists reason codes returned from callable services that give return code 0.

Table 87. ICSF Reason Codes for Return Code 0 (0)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
0 (0)	0 (0)	The call to the ICSF callable service or TSS verb was successfully processed. No error was encountered. **User action**: None.
4 (4)	2 (2)	The call to the ICSF callable service or TSS verb was successfully processed. A minor error was detected. A key used in the service did not have odd parity. This key could be one provided by you as a parameter or be one (perhaps of many) that was retrieved from the in-storage CKDS. **User action**: Refer to the reason code obtained when the key passed to this service was transformed into operational form using clear key import, multiple clear key import, key import, secure key import, or multiple secure key import callable services. Check if any of the services prepared an even parity key. If one of these service reported an even parity key, you need to know which key is affected. If none of these services identified an even parity key, then the even parity key detected was found on the CKDS. Report this to your administrator.
8 (8)	8 (8)	The key record read callable service attempted to read a NULL key record. The returned key token contains only binary zeros. **User action**: None required.
2710 (10000)	2710 (10000)	The call to the TSS verb was successfully processed. The keys in one or more key identifiers have been reenciphered from encipherment under the old master key to encipherment under the current master key. **User action**: If you obtained your operational token from a file, replace the token in the file with the token just returned from ICSF. Management of internal tokens is a user responsibility. Consider the possible case where the token for this call was fetched from a file, and where this reason code is ignored. For the next invocation of the service, the token will be fetched from the file again, and the service will give this reason code again. If this continues until the master key is changed again, then the next use of the internal token will fail.

ICSF Reason Codes for Return Code 4 (4)

Table 88 on page 337 lists reason codes returned from callable services that give return code 4.

Table 88. ICSF Reason Codes for Return Code 4 (4)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
0 (0)	0 (0)	Master key verification warning. There is a possible mismatch between the master key verification pattern in the CKDS and the system master key verification pattern. **User action**: Ensure that you specified the correct CKDS. If you specified the correct CKDS, check to see if the data set has been corrupted.
7D0 (2000)	014 (020)	The input text length was odd rather than even. The right nibble of the last byte is padded with X'00'. **User action**: None
BBA (3002)		The call to the CVV Verify callable service was successfully processed. However, the trial CVV that was supplied does not match the generated CVV. In addition, a key in the key identifier has been reenciphered. **User action**: See reason code 4000 for more details about the incorrect CVV. See reason code 10000 for more details about the key reencipherment.
BD4 (3028)	013 (019)	The call to the Encrypted PIN verify (PINVER) callable service was successfully processed. However, the trial PIN that was supplied does not match the PIN in the PIN block. **User action**: The PIN is incorrect. If you expected the reason code to be zero, check that you are using the correct key.
BD8 (3032)	013 (019)	This is a combination reason code value. The call to the Encrypted PIN verify (PINVER) callable service was successfully processed. However, the trial PIN that was supplied does not match the PIN in the PIN block. In addition, a key in a key identifier token has been reenciphered. **User action**: See reason code 3028 for more detail about the incorrect PIN. See reason code 10000 for more detail about the key reencipherment.
FA0 (4000)	01 (01)	The CVV did not verify. **User action**: Regenerate the CVV.
1F40 (8000)	01 (01)	The call to the MAC verification (MACVER) callable service was successfully processed. However, the trial MAC that you supplied does not match that of the message text. **User action**: The message text may have been modified, such that its contents cannot be trusted. If you expected the reason code to be zero, check that you are using the correct key. Check that all segments of the message were presented and in the correct sequence. Also check that the trial MAC corresponds to the message being authenticated.
1F44 (8004)	01 (01)	This is a combination reason code value. The call to the MAC verification (MACVER) callable service was successfully processed. However, the trial MAC that was supplied does not match the message text provided. In addition, a key in a key identifier token has been reenciphered. **User action**: See reason code 8000 for more detail about the incorrect MAC. See reason code 10000 for more detail about the key reencipherment.

Table 88. ICSF Reason Codes for Return Code 4 (4) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2328 (9000)	01 (01)	The call to the key test service processed successfully, but the key test pattern was not verified. **User action**: Investigate why the key failed. After determining this, you can reinstall or regenerate the key.
232C (9004)	01 (01)	This is a combination reason code value. The call to the key test service processed successfully, but the key test pattern was not verified. Also, the key token has been reenciphered. **User action**: Investigate why the key failed. After determining this, you can reinstall or regenerate the key.
2AF8 (11000)	1AD (429)	The digital signature verify ICSF callable service or TSS verb completed successfully but the supplied digital signature failed verification. **User action**: None
36B8 (14008)	01 (01)	The PKDS record failed the authentication test. **User action**: The record has changed since ICSF wrote it to the PKDS. The user action is application dependent.

ICSF Reason Codes for Return Code 8 (8)

Table 89 lists reason codes returned from callable services that give return code 8.

Most of these reason codes indicate that the call to the service was unsuccessful. No cryptographic processing took place. Therefore, no output parameters were filled. Exceptions to this are noted in the descriptions.

Table 89. ICSF Reason Codes for Return Code 8 (8)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
7D1 (2001)		TKE: DH generator is greater than the modulus.
7D2 (2002)		TKE: DH registers are not in a valid state for the requested operation.
7D3 (2003)		TKE: TSN does not match TSN in pending change buffer.
7D4 (2004)	019 (025)	A length parameter has an incorrect value. The value in the length parameter could have been zero (when a positive value was required) or a negative value. If the supplied value was positive, it could have been larger than your installation’s defined maximum, or for MDC generation with no padding, it could have been less than 16 or not an even multiple of 8. **User action**: Check the length you specified. If necessary, check your installation’s maximum length with your ICSF administrator. Correct the error.
7D5 (2005)		TKE: PCB data exceeds maximum data length.
7D8 (2008)	A4 (164)	Two parameters (perhaps the plaintext and ciphertext areas, or *text_in* and *text_out* areas) overlap each other. That is, some part of these two areas occupy the same address in memory. This condition cannot be processed. **User action**: Determine which two areas are responsible, and redefine their positions in memory.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
7D9 (2009)		TKE: ACI can not load both loads and profiles in one call.
7DA (2010)		TKE: ACI can only load one role or one profile at a time.
7DB (2011)		TKE: DH transport key algorithm match.
7DC (2012)	023 (035)	The *rule_array_count* parameter contains a number that is not valid. **User action**: Refer to the *rule_array_count* parameter described in this book under the appropriate callable service for the correct value.
7DD (2013)		TKE: Length of hash pattern for keypart is not valid for DH transport key algorithm specified.
7DE (2014)		TKE: PCB buffer is empty.
7E0 (2016)	021 (033), 09D (157)	The *rule_array* parameter contents are incorrect. **User action**: Refer to the *rule_array* parameter described in this book under the appropriate callable service for the correct value.
7E2 (2018)	021 (033)	The *form* parameter specified in the random number generate callable service should be ODD, EVEN, or RANDOM. One of these values was not supplied. **User action**: Change your parameter to use one of the required values for the *form* parameter.
7E3 (2019)		TKE: Signature in request CPRB did not verify.
7E4 (2020)		TKE: TSN in request CPRB is not valid.
7E8 (2024)	302 (770), 041 (065)	A reserved field in a parameter, probably a key identifier, has a value other than zero. **User action**: Key identifiers should not be changed by application programs for other uses. Review any processing you are performing on key identifiers and leave the reserved fields in them at zero.
7EB (2027)		TKE: DH transport key hash pattern doesn not match.
7EC (2028)	2C2 (706)	While deciphering ciphertext that had been created using a padding technique, it was found that the last byte of the plaintext did not contain a valid count of pad characters. Note that all cryptographic processing has taken place, and the *clear_text* parameter contains the deciphered text. **User action**: The *text_length* parameter was not reduced. Therefore, it contains the length of the base message, plus the length of the padding bytes and the count byte. Review how the message was padded before it was enciphered. The count byte that is not valid was created before the message's encipherment. You may need to check whether the ciphertext was not created using a padding scheme. Otherwise, check with the creator of the ciphertext on the method used to create it. You could also look at the plaintext to review the padding scheme used, if any.
7ED (2029)		TKE: Request data block hash does not match hash in CPRB.
7EE (2030)		TKE: DH supplied hash length is not correct.
7EF (2031)		Reply data block too large.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
7F0 (2032)		The *key_form, key_type_1*, and *key_type_2* parameters for the key generate callable service form a combination, a three-element string. This combination is checked against all valid combinations. Your combination was not found among this list. **User action**: Check the allowable combinations described for each parameter in Key Generate callable service and correct the appropriate parameter(s).
7F1 (2033)		TKE: Change type does not match PCB change type.
7F4 (2036)	0A5 (165)	The contents of a chaining vector passed to a callable service are not valid. If you called the MAC generation callable service, or the MDC generation callable service with a MIDDLE or LAST segmenting rule, the count field has a number that is not valid. If you called the MAC verification callable service, then this will have been a MIDDLE or LAST segmenting rule. **User action**: Check to ensure that the chaining vector is not modified by your program. The chaining vector returned by ICSF should only be used to process one message set, and not intermixed between alternating message sets. This means that if you receive and process two or more independent message streams, each should have its own chaining vector. Similarly, each message stream should have its own key identifier. If you use the same chaining vector and key identifier for alternating message streams, you will **not** get the correct processing performed.
7F6 (2038)		The caller must be in task mode, not SRB mode.
7F8 (2040)	0B5 (181), 03F (063), 09A (154)	This check is based on the first byte in the key identifier parameter. The key identifier provided is either an internal token, where an external or null token was required; or an external or null token, where an internal token was required. The token provided may be none of these, and, therefore, the parameter is not a key identifier at all. Another cause is specifying a *key_type* of IMP-PKA for a key in importable form. **User action**: Check the type of key identifier required and review what you have provided. Also check that your parameters are in the required sequence.
800 (2048)		The *key_form* is not valid for the *key_type* **User action**: Review the *key_form* and *key_type* parameters. For a *key_type* of IMP-PKA, the secure key import callable service supports only a *key_form* of OP.
802 (2050)		A UKPT keyword was specified, but there is an error in the *PIN_profile* key serial number. **User action**: Correct the PIN profile key serial number.
804 (2052)		A single-length key, passed to the secure key import callable service in the *clear_key* parameter, must be padded on the right with binary zeros. The fact that it is a single-length key is identified by the *key_form* parameter, which identifies the key as being DATA, MACGEN, MACVER, and so on. **User action**: If you are providing a single-length key, pad the parameter on the right with zeros. Alternatively, if you meant to pass a double-length key, correct the *key_form* parameter to a valid double-length key type.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
808 (2056)	029 (041)	The *key_form* parameter is neither IM nor OP. Most constants, these included, can be supplied in lower or uppercase. Note that this parameter is 4 bytes long, so the value IM or OP is not valid. They must be padded on the right with blanks. **User action**: Review the value provided and change it to IM or OP, as required.
80C (2060)	02B (043)	The *key_length* parameter passed to the key generate callable service holds a value that is not valid. **User action**: Review the value provided and change it as appropriate.
810 (2064)	0A0 (160)	The key_type and the key_length are not consistent. **User action**: Review the *key_type* parameter provided and match it with the *key_length* parameter.
814 (2068)		You supplied a key identifier or token to the key generate, key import, multiple secure key import, key export, or key record write callable service. This key identifier holds an importer or exporter key, and the NOCV bit is on in the token. Only programs running in supervisor state or in a system key (key 0–7) may provide a key identifier with this bit set on. Your program was not running in supervisor state or a system key. **User action**: Either use a different key identifier, or else run in supervisor state or a system key.
818 (2072)		A request was made to the key generate callable service to generate double-length keys of SINGLE effective length, in the IMEX form. This request is valid only if the *kek_key_identifier_1* parameter identifies a NOCV importer, and the caller (wrongly) supplies a CV importer. The combination of IMEX for the *key_form* parameter and a CV importer key-encrypting key can only be used for single-length keys. **User action**: Either use a key identifier that holds (or identifies) a NOCV importer, or specify a single-length key in the *key_type* parameter.
81C (2076)		A request was made to the key import callable service to import a single-length key. However, the right half of the key in the *source_key_identifier* parameter is not zeros. Therefore, it appears to identify the right half of a double-length key. This combination is not valid. This error does not occur if you are using the word TOKEN in the *key_type* parameter. **User action**: Check that you specified the value in the *key_type* parameter correctly, and that you are using the correct or corresponding *source_key_identifier* parameter.
824 (2084)		The key token is not valid for the CSNBTCK service. If the *source_key_identifier* is an external token, then the *kek_key_identifier* cannot be marked as CDMF. **User action**: Correct the appropriate key identifiers.
828 (2088)		The *origin_identifier* or *destination_identifier* you supplied is not a valid ASCII hexadecimal string. **User action**: Check that you specified a valid ASCII string for the *origin_identifier* or *destination_identifier* parameter.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
82C (2092)	079 (121)	The *source_key_identifier* or *inbound_key_identifier* you supplied in an ANSI X9.17 service is not a valid ASCII hexadecimal string. **User action**: Check that you specified a valid ASCII string for the *source_key_identifier* or *inbound_key_identifier* parameter.
830 (2096)	07A (122)	The *outbound_KEK_count* or *inbound_KEK_count* you supplied is not a valid ASCII hexadecimal string. **User action**: Check that you specified a valid ASCII hexadecimal string for the *outbound_KEK_count* or *inbound_KEK_count* parameter.
834 (2100)	2CB (715)	You supplied a *pad_character* that is not valid for a Transaction Security System compatibility parameter for which ICSF supports only one value; or, you supplied a KEY keyword and a non-zero *master_key_version_number* in the Key Token Build service; or, you supplied a non-zero regeneration data length for a DSS key in the PKA Generate service. **User action**: Check that you specified the valid value for the TSS compatibility parameter.
838 (2104)	02D (045)	An input character is not in the code table. **User action**: Correct the code table or the source text.
83C (2108)	02F (047)	An unused field must be binary zeros, and an unused key identifier field generally must be zeros. **User action**: Correct the parameter list.
840 (2112)		The length is incorrect for the key type. **User action**: Check the key length parameter. DATA keys may have a length of 8, 16, or 24. DATAXLAT and MAC keys must have a length of 8. All other keys should have a length of 16. Also check that the parameters are in the required sequence.
844 (2116)	021 (033)	Parameter contents or a parameter value is not correct. **User action**: Specify a valid value for the parameter.
BB9 (3001)		HCR7703 and higher systems - SET block decompose service was called with an encrypted OAEP block with a block contents identifier that indicates a PIN block is present. No PIN encrypting key was supplied to process the PIN block. The block contents identifier is returned in the *block_contents_identifier* parameter. OR HCRP220 or lower systems - A PKDS access has been attempted for a PKA token which exceeds the maximum PKA token size of 1024 bytes. This can occur if systems are sharing a PKDS and not all of the sharing systems support PKA tokens larger than 1024 bytes. **User action**: HCR7703 and higher systems - Supply a PIN encrypting key and resubmit the job. HCRP220 and lower systems - Check the key label supplied. The label must represent a PKDS record representing a PKA token of length less than or equal to 1024 bytes.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
BBC (3004)	064 (100)	A request was made to the Clear PIN generate or Encrypted PIN verify callable service, and the *PIN_length* parameter has a value outside the valid range. The valid range is from 4 to 16, inclusive. **User action**: Correct the value in the *PIN_length* parameter to be within the valid range from 4 to 16.
BBE (3006)		The UDX verb in the PCI Cryptographic Coprocessor is not authorized to be executed.
BC0 (3008)	065 (101)	A request was made to the Clear PIN generate callable service, and the *PIN_check_length* parameter has a value outside the valid range. The valid range is from 4 to 16, inclusive. **User action**: Correct the value in the *PIN_check_length* parameter to be within the valid range from 4 to 16.
BC4 (3012)	069 (105)	A request was made to the Clear PIN generate callable service to generate a VISA-PVV PIN, and the *trans_sec_parm* field has a value outside the valid range. The field being checked in the *trans_sec_parm* is the key index, in the 12th byte. This *trans_sec_parm* field is part of the *data_array* parameter. **User action**: Correct the value in the key index, held within the *trans_sec_parm* field in the *data_array* parameter, to hold a number from the valid range.
BC8 (3016)	06A (106)	A request was made to the Encrypted PIN translate or the Encrypted PIN verify callable service, and the PIN block value in the *input_PIN_profile* or *output_PIN_profile* parameter has a value that is not valid. **User action**: Correct the PIN block value.
BD0 (3024)	06B (107)	A request was made to the Encrypted PIN translate callable service and the format control value in the *input_PIN_profile* or *output_PIN_profile* parameter has a value that is not valid. The valid values are NONE or PBVC. **User action**: Correct the format control value to either NONE or PBVC.
BD4 (3028)	074 (116)	A request was made to the Clear PIN generate callable service. The clear_PIN supplied as part of the *data_array* parameter for an GBP-PINO request begins with a zero (0). This value is not valid. **User action**: Correct the clear_PIN value.
BDC (3036)	06F (111)	A request was made to the Encrypted PIN translate callable service. The *sequence_number* parameter was required, but was not the integer value 99999. **User action**: Specify the integer value 99999.
BE0 (3040)	06E (110)-PAN, 028 (040)-ser. code, 02A (042)-exp. date, 066 (102)-dec table, 067 (103)-val. table, 06C (198)-pad data	The PAN, expiration date, service code, decimalization table data, validation data, or pad data is not numeric (X'F0' through X'F9'). The parameter must be character representations of numerics or hexadecimal data. **User action**: Review the numeric parameters or fields required in the service that you called and change to the format and values required.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
FA0 (4000)	033 (051)	The encipher and decipher callable services sometime require text (plaintext or ciphertext) to have a length that is an exact multiple of 8 bytes. Padding schemes always create ciphertext with a length that is an exact multiple of 8. If you want to decipher ciphertext that was produced by a padding scheme, and the text length is not an exact multiple of 8, then an error has occurred. The CBC mode of enciphering requires a text length that is an exact multiple of 8. The ciphertext translate callable service cannot process ciphertext whose length is not an exact multiple of 8. **User action**: Review the requirements of the service you are using. Either adjust the text you are processing or use another process rule.
1388 (5000)		Target cryptographic module is not available in the configuration. **User action**: Correct the target cryptographic module parameter and resubmit.
138C (5004)		Format of the cryptographic request message is not valid. **User action**: Correct the request and resubmit it.
1390 (5008)		Length of the cryptographic request message is not valid. **User action**: Message length of request must be nonzero, a multiple of eight, and less than the system maximum. Correct the request and resubmit it.
2710 (10000)	01D (029), 00C (012), 02B (043)	A key identifier was passed to a service or token. It is checked in detail to ensure that it is a valid token, and that the fields within it are valid values. There is a token validation value (TVV) in the token, which is a non-cryptographic value. This value was again computed from the rest of the token, and compared to the stored TVV. If these two values are not the same, this reason code is returned. **User action**: The contents of the token have been altered because it was created by ICSF or TSS. Review your program to see how this could have been caused.
2714 (10004)	018 (024), 030 (048)	A key identifier was passed to a service. The master key verification pattern in the token shows that the key was created with a master key that is neither the current master key nor the old master key. Therefore, it cannot be reenciphered to the current master key. **User action**: Re-import the key from its importable form (if you have it in this form), or repeat the process you used to create the operational key form. If you cannot do one of these, you cannot repeat any previous cryptographic process that you performed with this token.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	**TSS Reason Code Hex (Decimal)**	**Description**
271C (10012)	01E (030)	A key label was supplied for a key identifier parameter. This label is the label of a key in the in-storage CKDS or the PKDS. Either the key could not be found, or a key record with that label and the specific type required by the ICSF callable service or TSS verb could not be found. For a retained key label, this error code is also returned if the key is not found in the PCI Cryptographic Coprocessor specified in the PKDS record. **User action**: Check with your administrator if you believe that this key should be in the in-storage CKDS or the PKDS. The administrator may be able to bring it into storage. If this key cannot be in storage, use a different label.
2720 (10016)	03D (061)	You specified a value for a *key_type* parameter that is not an ICSF-defined name. **User action**: Review the ICSF key types and use the appropriate one.
2724 (10020)	027 (039)	You specified the word TOKEN for a *key_type* parameter, but the corresponding key identifier, which implies the key type to use, has a value that is not valid in the control vector field. Therefore, a valid key type cannot be determined. **User action**: Review the value that you stored in the corresponding key identifier. Check that the value for *key_type* is obtained from the appropriate *key_identifier* parameter.
272C (10028)	027 (039)	Either the *left* half of the control vector in a key identifier (internal or external) equates to a key type that is not valid for the service you are using, or the value is not that of any ICSF control vector. For example, an exporter key-encrypting key is not valid in the key import callable service. **User action**: Determine which key identifier is in error and use the key identifier that is required by the service.
2730 (10032)	027 (039)	Either the *right* half of the control vector in a key identifier (internal or external) equates to a key type that is not valid for the service you are using, or the value is not that of any ICSF control vector. For example, an exporter key-encrypting key is not valid in the key import callable service. **User action**: Determine which key identifier is in error and use the key identifier that is required by the service.
2734 (10036)	027 (039)	Either the complete control vector (CV) in a key identifier (internal or external) equates to a key type that is not valid for the service you are using, or the value is not that of any ICSF control vector. The difference between this and reason codes 10028 and 10032 is that each half of the control vector is valid, but *as a combination*, the whole is not valid. For example, the left half of the control vector may be the importer key-encrypting key and the right half may be the input PIN-encrypting (IPINENC) key. **User action**: Determine which key identifier is in error and use the key identifier that is required by the service.
2738 (10040)	031 (049)	Key identifiers contain a version number. The version number in a supplied key identifier (internal or external) is inconsistent with one or more fields in the key identifier, making the key identifier unusable. **User action**: Use a token containing the required version number.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
273C (10044)	0B7 (183)	A cross-check of the control vector the key type implies has shown that it does not correspond with the control vector present in the supplied internal key identifier. **User action**: Change either the key type or key identifier.
2740 (10048)	03D (061)	The *key_type* parameter does not contain one of the valid types for the service or the keyword TOKEN. **User action**: Check the supplied parameter with the ICSF key types. If you supplied the keyword TOKEN, check that you have padded it on the right with blanks.
2744 (10052)	027 (039)	A null key identifier was supplied and the *key_type* parameter contained the word TOKEN. This combination of parameters is not valid. **User action**: Use either a null key identifier or the word TOKEN, not both.
2748 (10056)		You called the key import callable service. The importer key-encrypting key is a NOCV importer and you specified TOKEN for the *key_type* parameter. This combination is not valid. **User action**: Specify a value in the *key_type* parameter for the operational key form.
274C (10060)	03D (061)	You called the key export callable service. A label was supplied in the *key_identifier* parameter for the key to be exported and the *key_type* was TOKEN. This combination is not valid because the service needs a key type in order to retrieve a key from the CKDS. **User action**: Specify the type of key to be exported in the *key_type* parameter.
2754 (10068)	02F (047)	A flag in a key identifier indicates the master key verification pattern (MKVP) is not present in an internal key token. This setting is not valid. **User action**: Use a token containing the required flag values.
2758 (10072)	02F (047)	A flag in a key identifier indicates the encrypted key is not present in an external token. This setting is not valid. **User action**: Use a token containing the required flag values.
275C (10076)	02F (047)	A flag in a key identifier indicates the control vector is not present. This setting is not valid. **User action**: Use a token containing the required flag values.
2760 (10080)		An ICSF private flag in a key identifier has been set to a value that is not valid. **User action**: Use a token containing the required flag values. Do not modify ICSF or the reserved flags for your own use.
2768 (10088)	027 (039)	If you supplied a label in the *key_identifier* parameter, a record with the supplied label was found in the CKDS, but the key type (CV) is not valid for the service. If you supplied an internal key token for the *key_identifier* parameter, it contained a key type that is not valid. **User action**: Check with your ICSF administrator if you believe that this key should be in the in-storage CKDS. The administrator may be able to bring it into storage. If this key cannot be in storage, use a different label.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
276C (10092)		You supplied a source key that does not have odd parity and specified ENFORCE as the parity rule on the *rule_array* parameter for either the ANSI X9.17 key export, ANSI X9.17 key import, or ANSI X9.17 key translate callable service. **User action**: Either supply an ODD parity key or change the *rule_array* parameter to specify a parity rule of IGNORE.
2770 (10096)		The transport key you specified is a single-length key, which cannot be used to encrypt a double-length AKEK or (*KK). **User action**: Use a double-length AKEK for the transport key.
2774 (10100)		You specified a transport key that cannot be notarized and specified the keyword NOTARIZE in the *rule_array* parameter. The transport key may have already been partially notarized. **User action**: Use a transport key that allows notarization or change the *rule_array* parameter keyword to CPLT-NOT.
2778 (10104)		The AKEK you specified is either partially notarized or is a partial AKEK, which is not valid for this service. **User action**: Use a correct AKEK that is not partially notarized. A partially notarized key can be used as a transport key if you specify CPLT-NOT in the *rule_array* parameter.
277C (10108)		You did not supply a partial AKEK for the *key_identifier* parameter of the key part import service. **User action**: Correct the key_id parameter.
2780 (10112)		The transport key you specified has not been partially notarized and you have specified CPTL-NOT for the *rule_array* parameter. **User action**: Use a transport key that has been partially notarized or change the *rule_array* parameter.
2784 (10116)		You attempted to export an AKEK with a CCA key export service, which is not supported. **User action**: Use the ANSI X9.17 key export callable service (CSNAKEX).
2788 (10120)		The internal key token you supplied, or the key token that was retrieved by the label you supplied, contains a flag setting or data encryption algorithm bit that is not valid for this service. **User action**: Ensure that you supply a key token, or label, for a non-ANSI key type.
278C (10124)	027 (039)	The key identifier you supplied cannot be exported because there is a prohibit-export restriction on the key. **User action**: Use the correct key for the service.
2790 (10128)		The keyword you supplied in the *rule_array* parameter is not consistent or not valid with another parameter you specified. For example, the keyword SINGLE is not valid with the key type of EXPORTER in the key token build callable service. **User action**: Correct either the *rule_array* parameter or the other parameter.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2AF8 (11000)	048 (072)	The value specified for length parameter for a key token, key, or text field is not valid. **User action**: Correct the appropriate length field parameter.
2AFC (11004)	02F (047)	The hash value (of the secret quantities) in the private key section of the internal token failed validation. The values in the token are corrupted. You cannot use this key. **User action**: Recreate the token using the appropriate combination of the PKA key token build, PKA key generate, and PKA key import callable services.
2B00 (11008)	302 (770)	The public or private key values are not valid. (For example, the modulus or an exponent is zero.) You cannot use the key. **User action**: You may need to recreate the token using the PKA key token build or PKA key import callable service or regenerate the key values on another platform.
2B04 (11012)	02F (047)	The internal or external private key token contains flags that are not valid. **User action**: You may need to recreate the token using the PKA key token build or PKA key import callable service.
2B08 (11016)	02F (047)	The calculated hash of the public information in the PKA token does not match the hash in the private section of the token. The values in the token are corrupted. **User action**: Verify the public key section and the key name section of the token. If the token is still rejected, then you need to recreate the token using the appropriate combination of the PKA key token build, PKA key generate, and PKA key import callable services.
2B0C (11020)	030 (048)	The hash pattern of the PKA master key (SMK or KMMK) in the supplied internal PKA private key token does not match the current system's PKA master key. This indicates the system PKA master key has changed since the token was created. You cannot use the token. **User action**: Recreate the token using the appropriate combination of the PKA key token build, PKA key generate, and PKA key import callable services.
2B10 (11024)	02F (047)	The PKA tokens have incomplete values, for example, a PKA public key token without modulus. **User action**: Recreate the key.
2B14 (11028)	048 (072)	The modulus of the PKA key is too short for processing the hash or PKCS block. **User action**: Either use a PKA key with a larger modulus size, use a hash algorithm that generates a smaller hash (digital signature services), or specify a shorter DATA key size (symmetric key export, symmetric key generate).
2B18 (11032)	040 (064)	The supplied private key can be used only for digital signature. Key management services are disallowed. **User action**: Supply a key with key management enabled.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2B20 (11040)	042 (066)	The recovered encryption block was not a valid PKCS-1.2 or zero-pad format. (The format is verified according to the recovery method specified in the rule-array.) If the recovery method specified was PKCS-1.2, refer to PKCS-1.2 for the possible error in parsing the encryption block. **User action**: Ensure that the parameters passed to CSNDSYI are correct. Possible causes for this error are incorrect values for the RSA private key or incorrect values in the *RSA_enciphered_key* parameter, which must be formatted according to PKCS-1.2 or zero-pad rules when created.
2B24 (11044)	0B5 (181)	The first section of a supplied PKA token was not a private or public key section. **User action**: Recreate the key.
2B28 (11048)		The eyecatcher on the PKA internal private token is not valid. **User action**: Reimport the private token using the PKA key import callable service.
2B2C (11052)		An incorrect PKA token was supplied. The service requires a private key token. **User action**: Supply a PKA private key token as input.
2B30 (11056)		The input PKA token contains length fields that are not valid. **User action**: Recreate the key token.
2B38 (11064)	2CF (719)	The RSA-OAEP block did not verify after the decompose. The block type is incorrect (must be X'03'). **User action**: Recreate the RSA-OAEP block.
2B3C (11068)	2D1 (721)	The RSA-OAEP block did not verify after the decompose. The verification code is not correct (must be all zeros). **User action**: Recreate the RSA-OAEP block.
2B40 (11072)	2D0 (720)	The RSA-OAEP block did not verify after the decompose. The random number I is not correct (must be non-zero with the high-order bit equal to zero). **User action**: Recreate the RSA-OAEP block.
2B48 (11080)	041 (65), 2F8 (760)	The RSA public or private key specified a modulus length that is incorrect for this service. **User action**: Re-invoke the service with an RSA key with the proper modulus length.
2B4C (11084)		This service requires an RSA public key and the key identifier specified is not a public key. **User action**: Re-invoke the service with an RSA public key.
2B50 (11088)		This service requires an RSA private key that is for signature use only. **User action**: Re-invoke the service with a supported private key.
2B54 (11092)		There was an invalid subsection in the PKA token. **User action**: Correct the PKA token.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2B58 (11096)	040 (064)	This service requires an RSA private key that is for signature use. The specified key may be used for key management purposes only. **User action**: Re-invoke the service with a supported private key.
3E80 (16000)	Reason code 0, return code 308(776)	RACF failed your request to use this service. **User action**: Contact your ICSF or RACF administrator if you need this service.
3E84 (16004)	Reason code 1, return code 308(776)	RACF failed your request to use the key label. **User action**: Contact your ICSF or RACF administrator if you need this key.
3E8C (16012)		You requested the conversion service, but you are not running in an authorized state. **User action**: You must be running in supervisor state to use the conversion service. Contact your ICSF administrator.
3E90 (16016)	027 (039)	The input/output field contained a valid internal token with the NOCV bit on or encryption algorithm mark, but the key type was incorrect or did not match the type of the generated or imported key. Processing failed. **User action**: Correct the calling application.
3E94 (16020)		You requested dynamic CKDS update services for a system key, which is not allowed. **User action**: Correct the calling application.
3E98 (16024)	0B5 (181)	You called the key record write callable service, but the key token you supplied is not valid. **User action**: Check with your ICSF administrator if you believe that this key should be in the in-storage CKDS. The administrator may be able to bring it into storage. If this key cannot be in storage, use a different label.
3EA0 (16032)	020 (032)	You called the key record create callable service, but the *key_label* parameter syntax was incorrect. **User action**: Correct *key_label* syntax.
3EA4 (16036)	02C (044)	The key record create callable service requires that the key created not already exist in the CKDS. A key of the same label was found. **User action**: Make sure the application specifies the correct label. If the label is correct, contact your ICSF security administrator or system programmer.
3EA8 (16040)		Data in the PKDS record did not match the expected data. This occurs if the record does not contain a null PKA token and CHECK was specified. **User action**: If the record is to be overwritten regardless of its content, specify OVERLAY.

Table 89. ICSF Reason Codes for Return Code 8 (8) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
3EAC (16044)		One or more key labels specified as input to the PKA key generate or PKA key import service incorrectly refer to a retained private key. If generating a retained private key, this error may result from one of the following conditions: • The private key name of the retained private key being generated is the same as an existing PKDS record, but the PKDS record label was not specified as the input skeleton (source) key identifier. • The label specified in the *generated_key_token* parameter as the target for the retained private key was not the same as the private key name If generating or importing a non-retained key, this error occurs when the label specified as the target key specifies a retained private key. The retained private key cannot be over-written. **User action:** Make sure the application specifies the correct label. If the label is correct, contact your ICSF security administrator or system programmer.
3EB0 (16048)		Retained keys on the PKDS cannot be deleted or updated using the PKDS key record delete or PKDS key record write callable services, respectively. **User action:** Use the retained key delete callable service to delete retained keys.

ICSF Reason Codes for Return Code C (12)

Table 90 lists reason codes returned from callable services that give return code 12. These reason codes indicate that the call to the callable service was not successful. Either cryptographic processing did not take place, or the last cryptographic unit was switched offline. Therefore, no output parameters were filled.

Note: The higher-order halfword of the reason code field for return code C (12) may contain additional coding. See reason codes 273C and 2740 in the following table. For example, in the reason code 42738, the 4 is an SVC 99 error code and the 2738 is listed in the table below.

Table 90. ICSF Reason Codes for Return Code C (12)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
0 (0)		ICSF: ICSF is not available. Either ICSF was not started, or ICSF has started, but does not have access to any cryptographic units. Your request cannot be processed. **User action**: Check the availability of ICSF with your ICSF administrator.
4 (4)		The CKDS or PKDS management service you called is not available because it has been disallowed by the ICSF User Control Functions panel. **User action**: Contact the security administrator or system programmer to determine why the CKDS or PKDS management services have been disallowed.

Table 90. ICSF Reason Codes for Return Code C (12) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
8 (8)		The service or algorithm is not available on current hardware. Your request cannot be processed. **User action**: Correct the calling program or run on applicable hardware.
C (12)		The service that you called is unavailable because the installation exit for that service had previously failed. **User action**: Contact your ICSF administrator or system programmer.
10 (16)		A requested installation service routine could not be found. Your request was not processed. **User action**: Contact your ICSF administrator or system programmer.
1C (28)		Cryptographic asynchronous processor failed. **User action**: Contact your IBM support center.
20 (32)		Cryptographic asynchronous instruction was not executed. **User action**: Ensure cryptographic services are enabled.
32 (50)		An ICSF PKA service could not be performed because ICSF is being terminated. Any of the PKA services can issue this. **User action**: Review the reason code.
178C (6028)		ESTAE could not be established in common I/O routines. **User action**: Contact your system programmer or the IBM Support Center.
7D6 (2006)		TKE: PCB service error.
7D7 (2007)		TKE: Change type in PCB is not recognized.
7DF (2015)		Domain in CPRB not enabled by EMB mask.
7E1 (2017)		MKVP mismatch on Set MK.
7E5 (2021)		PCI Cryptographic Coprocessor adapter disabled.
7E9 (2025)		Enforcement mask error.
7F3 (2035)		Intrusion latch has been tripped. Services disabled.
7F5 (2037)		The domain specified is not valid.
7FB (2043)		OA certificate not found.
1790 (6032)		The dynamic allocation of the DASD copy of the CKDS or PKDS in use by ICSF failed. **User action**: Contact your ICSF security administrator or system programmer. The SVC 99 error code will be placed in the high-order halfword of the reason code field.
1794 (6036)		A dynamic deallocation error occurred when closing and deallocating a CKDS or PKDS. **User action**: Contact your security administrator or system programmer. The SVC 99 error code will be placed in the high-order halfword of the reason code field.

Table 90. ICSF Reason Codes for Return Code C (12) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2724 (10020)		A key retrieved from the in-storage CKDS failed the MAC verification (MACVER) check and is unusable. **User action**: Contact your ICSF administrator.
2728 (10024)		A key retrieved from the in-storage CKDS or a key to be written to the PKDS was rejected for use by the installation exit. **User action**: Contact your ICSF administrator or system programmer.
272C (10028)		You cannot use the secure key import or multiple secure key import callable services because the cryptographic unit is not enabled for processing. The cryptographic unit is not in special secure mode or is disabled in the environment control mask (ECM). **User action**: Contact your ICSF administrator (your administrator can enable the processing mode or the ECM).
2734 (10036)		More than one key with the same label was found in the CKDS or PKDS. This function requires a unique key per label. The probable cause may be the use of an incorrect label pointing to a key type that allows multiple keys per label. **User action**: Make sure the application specifies the correct label. If the label is correct, contact your ICSF security administrator or system programmer to verify the contents of the CKDS or PKDS.
273C (10044)		OPEN of the PKDS in use by ICSF failed. **User action**: Contact your ICSF security administrator or system programmer.
2740 (10048)	0C5 (197)	I/O error reading or writing to the DASD copy of the CKDS or PKDS in use by ICSF. **User action**: Contact your ICSF security administrator or system programmer. The RPL feedback code will be placed in the high-order halfword of the reason code field.
2744 (10052)		Automatic REFRESH to free storage in the linear section of the CKT failed. **User action**: Contact your ICSF security administrator or system programmer and request that a REFRESH be done.
274C (10060)		The I/O subtask terminated for an unexpected reason before completing the request. No dynamic CKDS or PKDS update services are possible at this point. **User action**: Contact your system programmer who can investigate the problem and restart the I/O subtask by stopping and restarting ICSF.
2B04 (11012)		This function is disabled in the environment control mask (ECM). **User action**: Contact your ICSF administrator.
2B08 (11016)	2FC (764)	The PKA master key is not in a valid state. **User action**: Contact your ICSF administrator.
2B0C (11020)	Return code 8, reason code 30F (783)	The modulus of the public or private key is larger than allowed and configured in the CCC. You cannot use this key on this system. **User action**: Regenerate the key with a smaller modulus size.

Table 90. ICSF Reason Codes for Return Code C (12) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2B10 (11024)		The system administrator has used the ICSF User Control Functions panel to disable the PKA functions. **User action**: Wait until administrator functions are complete and the PKA functions are again enabled.
2B18 (11032)		A CAMQ is valid for PKSC but not for PKA. **User action**: Contact your ICSF administrator.
2B1C (11036)		A PKDS is not available for processing. **User action**: Contact your ICSF administrator.
2B20 (11040)		The PKDS Control Record hash pattern is not valid. **User action**: Contact your ICSF administrator.
2B24 (11044)		The PKDS could not be accessed. **User action**: Contact your ICSF administrator.
2B28 (11048)		The PCI Cryptographic Coprocessor failed. **User action**: Contact your IBM support center.
2B2C (11052)		The specific PCI Cryptographic Coprocessor requested for service is temporarily unavailable. PKDS could not be accessed. The specific PCI Cryptographic Coprocessor may be attempting some recovery action. If recovery action is successful, the PCI Cryptographic Coprocessor will be made available. If the recovery action fails, the PCI Cryptographic Coprocessor will be made permanently unavailable. **User action**: Retry the function.
2B30 (11056)		The PCI Cryptographic Coprocessor failed. The response from the processor was incomplete. **User action**: Contact your IBM support center.
2B34 (11060)		The service could not be performed because the required PCI Cryptographic Coprocessor was not active. **User action**: If the service required a specific PCI Cryptographic Coprocessor, verify that the value specified is correct. Reissue the request when the required PCI Cryptographic Coprocessor is available.
2B38 (11064)		Service could not be performed because of a hardware error on the PCI Cryptographic Coprocessor.
2EDC (11996)		The Integrated Cryptographic Feature is not available for CKDS initialization because the cryptographic unit is not in special secure mode. **User action**: Contact your ICSF administrator.
2EE0 (12000)		You cannot use the Clear PIN generate callable service because the cryptographic unit is not enabled for processing. The cryptographic unit is not in special secure mode. **User action**: Contact your ICSF administrator who can enable the processing mode.

Table 90. ICSF Reason Codes for Return Code C (12) (continued)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
2EE4 (12004)		An error occurred in a latch manager call. **User action**: Contact your ICSF security administrator or system programmer.
8CB4 (36020)		A refresh of the CKDS failed because the DASD copy of the CKDS is enciphered under the wrong master key. This may have resulted from an automatic refresh during processing of the key record create callable service. **User action**: Contact your ICSF administrator.

ICSF Reason Codes for Return Code 10 (16)

Table 91 lists reason codes returned from callable services that give return code 16.

Table 91. ICSF Reason Codes for Return Code 10 (16)

ICSF Reason Code Hex (Decimal)	TSS Reason Code Hex (Decimal)	Description
4 (4)		ICSF: Your call to an ICSF callable service resulted in an abnormal ending. The request parameter block failed consistency checking. **User action**: Contact your system programmer or the IBM Support Center.

Transaction Security System Return Codes and Reason Codes

The following section describes the return codes and reason codes that are returned after a call to a Transaction Security System verb has been completed. It also lists TSS to ICSF return codes and reason codes. Each TSS return code returns unique reason codes to your application program. The reason codes associated with each return code are described in the following sections. The reason code tables present the TSS hexadecimal code followed by the decimal code in parenthesis. If there is a 1-to-1 mapping, the codes will be converted. If there is not a map to ICSF, the column will be blank. If there are multiple mappings, they will be listed as reference only and will not be converted.

TSS Reason Codes for Return Code 0 (0)

Table 92 lists reason codes returned from TSS verbs that give return code 0 and any corresponding ICSF reason codes.

Table 92. TSS Reason Codes for Return Code 0 (0)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
0 (0)	0 (0)	The call to the ICSF callable service or TSS verb was successfully processed. No error was encountered. **User action**: None.

Table 92. TSS Reason Codes for Return Code 0 (0) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
2 (2)	4 (4)	The call to the ICSF callable service or TSS verb was successfully processed. A minor error was detected. A key used in the service was shown to have even parity. This key could be one provided by you as a parameter or be one (perhaps of many) that was retrieved from the in-storage CKDS. **User action**: Refer to the reason code obtained when the key passed to this service was transformed into operational form using clear key import, multiple clear key import, key import, secure key import, or multiple secure key import callable services. Check if any of the services prepared an even parity key. If one of these service reported an even parity key, you need to know which key is affected. If none of these services identified an even parity key, then the even parity key detected was found on the CKDS. Report this to your administrator.
8 (8)	8 (8)	The key record read callable service attempted to read a NULL key record. The returned key token contains only binary zeros. **User action**: None required.
2710 (10000)	2710 (10000)	The call to the TSS verb was successfully processed. The keys in one or more key identifiers have been reenciphered from encipherment under the old master key to encipherment under the current master key. **User action**: If you obtained your operational token from a file, replace the token in the file with the token just returned from ICSF. Management of internal tokens is a user responsibility. Consider the possible case where the token for this call was fetched from a file, and where this reason code is ignored. For the next invocation of the service, the token will be fetched from the file again, and the service will give this reason code again. If this continues until the master key is changed again, then the next use of the internal token will fail.
2711 (10001)		The call to the TSS verb was successfully processed. The keys in one or more key identifiers were encrypted under the old master key. The verb was unable to reencipher the key.

TSS Reason Codes for Return Code 4 (4)

Table 93 lists reason codes returned from TSS verbs that give return code 4 and any corresponding ICSF reason codes.

Table 93. TSS Reason Codes for Return Code 4 (4)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
01 (01)	FA0 (4000), 1F40 (8000), 1F44 (8004), 2328 (9000), 232C (9004), 36B8 (14008)	The verification test failed.

Table 93. TSS Reason Codes for Return Code 4 (4) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
013 (019)	BD4 (3028)	The call to the Encrypted PIN verify (PINVER) callable service was successfully processed. However, the trial PIN that was supplied does not match the PIN in the PIN block. **User action**: The PIN is incorrect. If you expected the reason code to be zero, check that you are using the correct key.
014 (020)	7D0 (2000)	The input text length was odd rather than even. The right nibble of the last byte is padded with X'00'. **User action**: None
0A6 (166)		The control vector is not valid because of parity bits, anti-variant bits, inconsistent KEK bits, or because bits 59 to 62 are not zero.
0B3 (179)		The control vector keywords that are in the rule array are ignored.
1AD (429)	2AF8 (11000)	The digital signature verify ICSF callable service or TSS verb completed successfully but the supplied digital signature failed verification. **User action**: None

Reason Codes for Return Code 8 (8)

Table 94 lists reason codes returned from TSS verbs that give return code 8 and any corresponding ICSF reason codes.

Table 94. TSS Reason Codes for Return Code 8 (8)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
00C (012)	2710 (10000)	A key identifier was passed to a service or token. It is checked in detail to ensure that it is a valid token, and that the fields within it are valid values. There is a token validation value (TVV) in the token, which is a non-cryptographic value. This value was again computed from the rest of the token, and compared to the stored TVV. If these two values are not the same, this reason code is returned. **User action**: The contents of the token have been altered because it was created by ICSF or TSS. Review your program to see how this could have been caused.
016 (022)		The ID number in the request field is not valid. Missing section in PKA token.
017 (023)		An access to the data area was outside the data area boundary.
018 (024)	2714 (10004)	The master key verification pattern is not valid.
01D (029)	2710 (10000)	A key identifier was passed to a service or token. It is checked in detail to ensure that it is a valid token, and that the fields within it are valid values. There is a token validation value (TVV) in the token, which is a non-cryptographic value. This value was again computed from the rest of the token, and compared to the stored TVV. If these two values are not the same, this reason code is returned. **User action**: The contents of the token have been altered because it was created by ICSF or TSS. Review your program to see how this could have been caused.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
01E (030)	271C (10012)	A key label was supplied for a key identifier parameter. This label is the label of a key in the in-storage CKDS or the PKDS. Either the key could not be found, or a key record with that label and the specific type required by the ICSF callable service or TSS verb could not be found. For a retained key label, this error code is also returned if the key is not found in the PCI Cryptographic Coprocessor specified in the PKDS record. **User action**: Check with your administrator if you believe that this key should be in the in-storage CKDS or the PKDS. The administrator may be able to bring it into storage. If this key cannot be in storage, use a different label.
01F (031)	272C (10028)	The control vector did not specify a DATA key.
020 (032)	3EA0 (16032)	You called the key record create callable service, but the *key_label* parameter syntax was incorrect. **User action**: Correct *key_label* syntax.
021 (033)	7E0 (2016)	The *rule_array* parameter contents or a parameter value is not correct. **User action**: Refer to the *rule_array* parameter described in this book under the appropriate callable service for the correct value.
022 (034)	7E0 (2016)	A rule array keyword combination is not valid.
023 (035)	7DC (2012)	The *rule_array_count* parameter contains a number that is not valid. **User action**: Refer to the *rule_array_count* parameter described in this book under the appropriate callable service for the correct value.
027 (039)	272C (10028), 2730 (10032), 2734 (10036), 2744 (10052), 2768 (10088), 278C (10124), 3E90 (16016), 2724 (10020)	A control vector violation occurred.
028 (040)	BE0 (3040)	The service code does not contain numerical character data.
029 (041)	808 (2056)	The *key_form* parameter is neither IM nor OP. Most constants, these included, can be supplied in lower or uppercase. Note that this parameter is 4 bytes long, so the value IM or OP is not valid. They must be padded on the right with blanks. **User action**: Review the value provided and change it to IM or OP, as required.
02A (042)- expiration date	BE0 (3040)	The expiration date is not numeric (X'F0' through X'F9'). The parameter must be character representations of numerics or hexadecimal data. **User action**: Review the numeric parameters or fields required in the service that you called and change to the format and values required.
02B (043)	80C (2060)	The *key_length* parameter passed to the key generate callable service holds a value that is not valid. **User action**: Review the value provided and change it as appropriate.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
02C (044)		The key record create callable service requires that the key created not already exist in the CKDS. A key of the same label was found. **User action**: Make sure the application specifies the correct label. If the label is correct, contact your ICSF security administrator or system programmer.
02D (045)		An input character is not in the code table. **User action**: Correct the code table or the source text.
02F (047)	83C (2108), 2754 (10068), 2758 (10072), 275C (10076), 2AFC (11004), 2B04 (11012), 2B08 (11016), 2B10 (11024)	A source key token is unusable because it contains data that is not valid or undefined.
030 (048)	2714 (10004), 2B0C (11020)	One or more keys has a master key verification pattern that is not valid.
031 (049)	2738 (10040)	Key identifiers contain a version number. The version number in a supplied key identifier (internal or external) is inconsistent with one or more fields in the key identifier, making the key identifier unusable. **User action**: Use a token containing the required version number.
03D (061)	2720 (10016), 2740 (10048), 274C (10060)	The keyword supplied with the *key_type* parameter is not valid.
03E (062)	271C (10012)	The source key was not found.
03F (063)	7F8 (2040)	This check is based on the first byte in the key identifier parameter. The key identifier provided is either an internal token, where an external or null token was required; or an external or null token, where an internal token was required. The token provided may be none of these, and, therefore, the parameter is not a key identifier at all. Another cause is specifying a *key_type* of IMP-PKA for a key in importable form. **User action**: Check the type of key identifier required and review what you have provided. Also check that your parameters are in the required sequence.
040 (064)	2B18 (11032), 2B58 (11096)	The supplied private key can be used only for digital signature. Key management services are disallowed. **User action**: Supply a key with key management enabled.
041 (065)	7E8 (2024)	The RSA public or private key specified a modulus length that is incorrect for this service. **User action**: Re-invoke the service with an RSA key with the proper modulus length.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
042 (066)	2B20 (11040)	The recovered encryption block was not a valid PKCS-1.2 or zero-pad format. (The format is verified according to the recovery method specified in the rule-array.) If the recovery method specified was PKCS-1.2, refer to PKCS-1.2 for the possible error in parsing the encryption block. **User action**: Ensure that the parameters passed to CSNDSYI are correct. Possible causes for this error are incorrect values for the RSA private key or incorrect values in the *RSA_enciphered_key* parameter, which must be formatted according to PKCS-1.2 or zero-pad rules when created.
043 (067)		DES or RSA encryption failed.
044 (068)		DES or RSA decryption failed.
048 (072)	2AF8 (11000), 2B14 (11028)	The value specified for length parameter for a key token, key, or text field is not valid. **User action**: Correct the appropriate length field parameter.
05A (090)		Access is denied for this verb. The authorization level is either too low or is not identical.
064 (100)	BBC (3004)	The PIN length is not valid.
065 (101)	BC0 (3008)	The PIN check length is not valid. It must be in the range from 4 to the PIN length inclusive.
066 (102)	BE0 (3040)	The value of the decimalization table is not valid.
067 (103)	BE0 (3040)	The value of the validation data is not valid.
068 (104)	BE0 (3040)	The value of the customer-selected PIN is not valid, or the PIN length does not match the value specified.
069 (105)	BE0 (3040)	The value of the *transaction_security_parameter* is not valid.
06A (106)	BC8 (3016)	The PIN block format keyword is not valid.
06B (107)	BD0 (3024)	The format control keyword is not valid.
06C (108)	BC8 (3016)	The value of the PAD data is not valid.
06D (109)		The extraction method keyword is not valid.
06E (110)	BE0 (3040)	The value of the PAN data is not numeric character data.
074 (116)	BBC (3004)	The clear PIN value is not valid.
079 (121)		The *source_key_identifier* or *inbound_key_identifier* you supplied in an ANSI X9.17 service is not a valid ASCII hexadecimal string. **User action**: Check that you specified a valid ASCII string for the *source_key_identifier* or *inbound_key_identifier* parameter.
09A (154)	7F8 (2040)	This check is based on the first byte in the key identifier parameter. The key identifier provided is either an internal token, where an external or null token was required; or an external or null token, where an internal token was required. The token provided may be none of these, and, therefore, the parameter is not a key identifier at all. Another cause is specifying a *key_type* of IMP-PKA for a key in importable form. **User action**: Check the type of key identifier required and review what you have provided. Also check that your parameters are in the required sequence.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
09B (155)		The value that the *generated_key_identifier* parameter specifies is not valid,or it is not consistent with the value that the *key_form* parameter specifies.
09C (156)	2790 (10128)	A keyword is not valid with the specified parameters.
09D (157)	7E0 (2016)	The key-token type is not specified in the rule array.
0A0 (160)		The key type and the key length are not consistent.
0A5 (165)	7F4 (2036)	The contents of a chaining vector passed to a callable service are not valid. If you called the MAC generation callable service, or the MDC generation callable service with a MIDDLE or LAST segmenting rule, the count field has a number that is not valid. If you called the MAC verification callable service, then this will have been a MIDDLE or LAST segmenting rule. **User action**: Check to ensure that the chaining vector is not modified by your program. The chaining vector returned by ICSF should only be used to process one message set, and not intermixed between alternating message sets. This means that if you receive and process two or more independent message streams, each should have its own chaining vector. Similarly, each message stream should have its own key identifier. If you use the same chaining vector and key identifier for alternating message streams, you will **not** get the correct processing performed.
0B5 (181)		This check is based on the first byte in the key identifier parameter. The key identifier provided is either an internal token, where an external or null token was required; or an external or null token, where an internal token was required. The token provided may be none of these, and, therefore, the parameter is not a key identifier at all. Another cause is specifying a *key_type* of IMP-PKA for a key in importable form. **User action**: Check the type of key identifier required and review what you have provided. Also check that your parameters are in the required sequence.
0B7 (183)	273C (10044)	A cross-check of the control vector the key type implies has shown that it does not correspond with the control vector present in the supplied internal key identifier. **User action**: Change either the key type or key identifier.
0B8 (184)		An input pointer is null.
0CC (204)		A memory allocation failed.
154 (340)		One of the input control vectors has odd parity.
157 (343)		Either the data block or the buffer for the block is too small.
159 (345)		Insufficient storage space exists for the data in the data block buffer.
15A (346)		The requested command is not valid in the current state of the cryptographic hardware component.
176 (374)	7D4 (2004), 7E0 (2016)	Less data was supplied than expected or less data exists than was requested.
181 (385)		The cryptographic hardware component reported that the data passed as part of the command is not valid for that command.
197 (407)	BC8 (3016)	A PIN block consistency check error occurred.
25D (605)		The number of output bytes is greater than the number that is permitted.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
2BF (703)		A new master key value was found to be one of the weak DES keys.
2C0 (704)		The new master key would have the same master key verification pattern as the current master key.
2C1 (705)		The same key-encrypting key was specified for both exporter keys.
2C2 (706)	7EC (2028)	While deciphering ciphertext that had been created using a padding technique, it was found that the last byte of the plaintext did not contain a valid count of pad characters. Note that all cryptographic processing has taken place, and the *clear_text* parameter contains the deciphered text. **User action**: The *text_length* parameter was not reduced. Therefore, it contains the length of the base message, plus the length of the padding bytes and the count byte. Review how the message was padded before it was enciphered. The count byte that is not valid was created before the message's encipherment. You may need to check whether the ciphertext was not created using a padding scheme. Otherwise, check with the creator of the ciphertext on the method used to create it. You could also look at the plaintext to review the padding scheme used, if any.
2C3 (707)		The master key registers are not in the state required for the requested function.
2CA (714)	844 (2116)	A reserved parameter was not a null pointer or an expected value.
2CB (715)	834 (2100)	A non-zero value was specified for a field that must be zero.
2CF (719)	2B38 (11064)	The RSA-OAEP block did not verify after the decompose. The block type is incorrect (must be X'03'). **User action**: Recreate the RSA-OEAP block.
2D0 (720)	2B40 (11072)	The RSA-OAEP block did not verify after the decompose. The random number I is not correct (must be non-zero with the high-order bit equal to zero). **User action**: Recreate the RSA-OEAP block.
2D1 (721)	2B3C (11068)	The RSA-OAEP block did not verify after the decompose. The verification code is not correct (must be all zeros). **User action**: Recreate the RSA-OEAP block.
2F8 (760)	2B48 (11080)	The RSA public or private key specified a modulus length that is incorrect for this service. **User action**: Re-invoke the service with an RSA key with the proper modulus length.
302 (770)	2B00 (11008)	A reserved field in a parameter, probably a key identifier, has a value other than zero. **User action**: Key identifiers should not be changed by application programs for other uses. Review any processing you are performing on key identifiers and leave the reserved fields in them at zero.
30F (783)	Return code 12, reason code 2B0C (11020)	The command is not permitted by Function-Control-Vector value.

Table 94. TSS Reason Codes for Return Code 8 (8) (continued)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
401 (1025)		Registered public key or retained private key name already exists.
405 (1029)		There is an error in the Environment Identification data.
41A (1050)		A KEK RSA-enciphered at this node (EID) cannot be imported at this same node.
7DF (2015)		An error occurred in the Domain Manager.
802 (2050)		A UKPT keyword was specified, but there is an error in the *PIN_profile* key serial number. **User action**: Correct the PIN profile key serial number.

TSS Reason Codes for Return Code C (12)

Table 95 lists reason codes returned from TSS verbs that give return code C and any corresponding ICSF reason codes.

Table 95. TSS Reason Codes for Return Code C (12)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
2FC (764)		The PKA master key is not in a valid state. **User action**: Contact your ICSF administrator.

TSS Reason Codes for Return Code 10 (16)

Table 96 lists reason codes returned from TSS verbs that give return code 10 and any corresponding ICSF reason codes. These error codes will result in an ICSF 18F abend with reason code X'50'. The caller will get return code 16, reason code 4.

Table 96. TSS Reason Codes for Return Code 10 (16)

TSS Reason Code Hex (Decimal)	ICSF Reason Code Hex (Decimal)	Description
150 (336)	4 (4)	An error occurred in the cryptographic hardware component.
22C (556)	4 (4)	The request parameter block failed consistency checking.
2C4 (708)	4 (4)	Inconsistent data was returned from the cryptographic engine.
2C5 (709)	4 (4)	Cryptographic engine internal error; could not access the master key data.
2C8 (712)	4 (4)	An unexpected error occurred in the Master Key manager.

Appendix B. Key Token Formats

For debugging purposes, this appendix provides the formats for DES internal, external, and null key tokens and for PKA key tokens.

Format of the DES Internal Key Token

Table 97 shows the format for a DES internal key token.

Table 97. Internal Key Token Format

Bytes	Description
0	X'01' (flag indicating this is an internal key token)
1–3	Implementation-dependent bytes (X'000000' for ICSF)
4	Key token version number (X'00' or X'01')
5	Reserved (X'00')
6	Flag byte **Bit** **Meaning When Set On** **0** Encrypted key and master key verification pattern (MKVP) are present. **1** Control vector (CV) value in this token has been applied to the key. **2** Key is used for no control vector (NOCV) processing. Valid for transport keys only. **3** Key is an ANSI key-encrypting key (AKEK). **4** AKEK is a double-length key (16 bytes). **Note:** When bit 3 is on and bit 4 is off, AKEK is a single-length key (8 bytes). **5** AKEK is partially notarized. **6** Key is an ANSI partial key. **7** Export prohibited.
7	Reserved (X'00')
8–15	Master key verification pattern (MKVP)
16–23	A single-length key, the left half of a double-length key, or Part A of a triple-length key. The value is encrypted under the master key.
24–31	X'0000000000000000' if a single-length key, or the right half of a double-length operational key, or Part B of a triple-length operational key. The right half of the double-length key or Part B of the triple-length key is encrypted under the master key.
32–39	The control vector (CV) for a single-length key or the left half of the control vector for a double-length key.
40–47	X'0000000000000000' if a single-length key or the right half of the control vector for a double-length operational key.
48–55	X'0000000000000000' if a single-length key or double-length key, or Part C of a triple-length operational key. Part C of a triple-length key is encrypted under the master key.
56-58	Reserved (X'000000')
59 bits 0 and 1	**B'10'** Indicates CDMF DATA or KEK. **B'00'** Indicates DES for DATA keys or the system default algorithm for a KEK. **B'01'** Indicates DES for a KEK.
59 bits 2 and 3	**B'00'** Indicates single-length key (version 0 only). **B'01'** Indicates double-length key (version 1 only). **B'10'** Indicates triple-length key (version 1 only).

Table 97. Internal Key Token Format (continued)

Bytes	Description
59 bits 4 –7	B'0000'
60–63	Token validation value (TVV).

Note: A key token stored in the CKDS will not have an MKVP or TVV. Before such a key token is used, the MKVP is copied from the CKDS header record and the TVV is calculated and placed in the token. See “Token Validation Value” for more information.

Token Validation Value

ICSF uses the *token validation value (TVV)* to verify that a token is valid. The TVV prevents a key token that is not valid or that is overlaid from being accepted by ICSF. It provides a checksum to detect a corruption in the key token.

When an ICSF callable service generates a key token, it generates a TVV and stores the TVV in bytes 60-63 of the key token. When an application program passes a key token to a callable service, ICSF checks the TVV. To generate the TVV, ICSF performs a twos complement ADD operation (ignoring carries and overflow) on the key token, operating on four bytes at a time, starting with bytes 0-3 and ending with bytes 56-59.

DES External Key Token

Table 98 on page 367 shows the format for a DES external key token.

Table 98. Format of External Key Tokens

Bytes	**Description**
0	X'02' (flag indicating an external key token)
1	Reserved (X'00')
2–3	Implementation-dependent bytes (X'0000' for ICSF)
4	Key token version number (X'00' or X'01')
5	Reserved (X'00')
6	Flag byte **Bit** **Meaning When Set On** **0** Encrypted key is present. **1** Control vector (CV) value has been applied to the key. Other bits are reserved and are binary zeros.
7	Reserved (X'00')
8–15	Reserved (X'0000000000000000')
16–23	Single-length key or left half of a double-length key, or Part A of a triple-length key. The value is encrypted under a transport key.
24–31	X'0000000000000000' if a single-length key or right half of a double-length key, or Part B of a triple-length key. The right half of a double-length key or Part B of a triple-length key is encrypted under a transport (key-encrypting key) for export or import.
32–39	Control vector (CV) for single-length key or left half of CV for double-length key
40–47	X'0000000000000000' if single-length key or right half of CV for double-length key
48–55	X'0000000000000000' if a single-length key, double-length key, or Part C of a triple-length key.
56–58	Reserved (X'000000')
59 bits 0 and 1	B'00'
59 bits 2 and 3	**B'00'** Indicates single-length key (version 0 only). **B'01'** Indicates double-length key (version 1 only). **B'10'** Indicates triple-length key (version 1 only).
59 bits 4–7	B'0000'
60-63	Token validation value (see “Token Validation Value” on page 366 for a description).

DES Null Key Token

Table 99 on page 368 shows the format for a DES null key token.

Table 99. Format of Null Key Tokens

Bytes	Description
0	X'00' (flag indicating this is a null key token).
1–15	Reserved (set to binary zeros).
16–23	Single-length encrypted key, or left half of double-length encrypted key, or Part A of triple-length encrypted key.
24–31	X'0000000000000000' if a single-length encrypted key, the right half of double-length encrypted key, or Part B of triple-length encrypted key.
32–39	X'0000000000000000' if a single-length encrypted key or double-length encrypted key.
40–47	Reserved (set to binary zeros).
48–55	Part C of a triple-length encrypted key.
56–63	Reserved (set to binary zeros).

Format of the RSA Public Key Token

An RSA public key token contains the following sections:

- A required token header, starting with the token identifier X'1E'
- A required RSA public key section, starting with the section identifier X'04'

Table 100 presents the format of an RSA public key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format).

Table 100. RSA Public Key Token

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1E' indicates an external token.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be zero.
RSA Public Key Section (required)		
000	001	X'04', section identifier, RSA public key.
001	001	X'00', version.
002	002	Section length, 12+xxx+yyy.
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, “xxx”.
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, “yyy”.
012	xxx	Public key exponent (this is generally a 1-, 3-, or 64- to 256-byte quantity), e. e must be odd and 1<e<n. (Frequently, the value of e is 2.) ^{16}+1
12+xxx	yyy	Modulus, n.

Format of the DSS Public Key Token

A DSS public key token contains the following sections:

- A required token header, starting with the token identifier X'1E'
- A required DSS public key section, starting with the section identifier X'03'

Table 101 presents the format of a DSS public key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format).

Table 101. DSS Public Key Token

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1E' indicates an external token.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be zero.
DSS Public Key Section (required)		
000	001	X'03', section identifier, DSS public key.
001	001	X'00', version.
002	002	Section length, 14+ppp+qqq+ggg+yyy.
004	002	Size of p in bits. The size of p must be one of: 512, 576, 640, 704, 768, 832, 896, 960, or 1024.
006	002	Size of the p field in bytes, “ppp”.
008	002	Size of the q field in bytes, “qqq”.
010	002	Size of the g field in bytes, “ggg”.
012	002	Size of the y field in bytes, “yyy”.
014	ppp	Prime modulus (large public modulus), p.
014 +ppp	qqq	Prime divisor (small public modulus), q. $2^{159}<q<2^{160}$.
014 +ppp +qqq	ggg	Public key generator, g.
014 +ppp +qqq +ggg	yyy	Public key, y. $y=g^x \bmod(p)$; $1<y<p$.

Format of RSA Private External Key Tokens

An RSA private external key token contains the following sections:

- A required PKA token header starting with the token identifier X'1E'
- A required RSA private key section starting with one of the following section identifiers:
  - X'02' which indicates a modulus-exponent form RSA private key section (not optimized) with modulus length of up to 1024 bits for use with the Cryptographic Coprocessor Feature or the PCI Cryptographic Coprocessor.
  - X'08' which indicates an optimized Chinese Remainder Theorem form private key section with modulus bit length of up to 2048 bits for use with the PCI Cryptographic Coprocessor
- A required RSA public key section, starting with the section identifier X'04'
- An optional private key name section, starting with the section identifier X'10'

Table 102 presents the basic record format of an RSA private external key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-justified and padded with zeros to the left.

Table 102. RSA Private External Key Token Basic Record Format

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1E' indicates an external token. The private key is either in cleartext or enciphered with a transport key-encrypting key.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be zero.
RSA Private Key Section (required) • For 1024-bit Modulus-Exponent form refer to "RSA Private Key Token, 1024-bit Modulus-Exponent External Form" • For 2048-bit Chinese Remainder Theorem form refer to "RSA Private Key Token, 2048-bit Chinese Remainder Theorem External Form" on page 371		
RSA Public Key Section (required)		
000	001	X'04', section identifier, RSA public key.
001	001	X'00', version.
002	002	Section length, 12+xxx.
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, "xxx".
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, which is zero for a private token. **Note:** In an RSA private key token, this field should be zero. The RSA private key section contains the modulus.
012	xxx	Public key exponent, e (this is generally a 1-, 3-, or 64- to 256-byte quantity). e must be odd and 1<e<n. (Frequently, the value of e is 2^{16}+1 (=65,537).
Private Key Name (optional)		
000	001	X'10', section identifier, private key name.
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-justified, padded with space characters (X'20'). An access control system can use the private key name to verify that the calling application is entitled to use the key.

RSA Private Key Token, 1024-bit Modulus-Exponent External Form

This RSA private key token and the external X'02' token is supported on the Cryptographic Coprocessor Feature and PCI Cryptographic Coprocessor.

Table 103. RSA Private Key Token, 1024-bit Modulus-Exponent External Format

Offset (Dec)	Number of Bytes	Description
000	001	X'02', section identifier, RSA private key, modulus-exponent format (RSA-PRIV)

Table 103. RSA Private Key Token, 1024-bit Modulus-Exponent External Format (continued)

Offset (Dec)	Number of Bytes	Description
001	001	X'00', version.
002	002	Length of the RSA private key section X'016C' (364 decimal).
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'00'** Unencrypted RSA private key subsection identifier. **X'82'** Encrypted RSA private key subsection identifier.
029	001	Reserved, binary zero.
030	020	SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X'00'.
050	004	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zero.
054	006	Reserved; set to binary zero.
060	024	Reserved; set to binary zero.
084	Start of the optionally-encrypted secure subsection.	
084	024	Random number, confounder.
108	128	Private-key exponent, d. $d=e^{-1} \bmod((p-1)(q-1))$, and $1<d<n$ where e is the public exponent.
	End of the optionally-encrypted subsection; the confounder field and the private-key exponent field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate that the private key is enciphered. They are enciphered under a double-length transport key using the ede2 algorithm.	
236	128	Modulus, n. n=pq where p and q are prime and $1<n<2^{1024}$.

RSA Private Key Token, 2048-bit Chinese Remainder Theorem External Form

This RSA private key token is supported on the PCI Cryptographic Coprocessor.

Table 104. RSA Private Key Token, 2048-bit Chinese Remainder Theorem External Format

Offset (Dec)	Number of Bytes	Description
000	001	X'08', section identifier, RSA private key, CRT format (RSA-CRT)
001	001	X'00', version.
002	002	Length of the RSA private-key section, 132 + ppp + qqq + rrr + sss + uuu + xxx + nnn.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the end of the modulus.
024	004	Reserved; set to binary zero.

Table 104. RSA Private Key Token, 2048-bit Chinese Remainder Theorem External Format (continued)

Offset (Dec)	Number of Bytes	Description
028	001	Key format and security: **X'40'** Unencrypted RSA private-key subsection identifier, Chinese Remainder form. **X'42'** Encrypted RSA private-key subsection identifier, Chinese Remainder form.
029	001	Reserved; set to binary zero.
030	020	SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, then 20 bytes of X'00'.
050	004	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zero.
054	002	Length of prime number, p, in bytes: ppp.
056	002	Length of prime number, q, in bytes: qqq.
058	002	Length of d_p, in bytes: rrr.
060	002	Length of d_q, in bytes: sss.
062	002	Length of U, in bytes: uuu.
064	002	Length of modulus, n, in bytes: nnn.
066	004	Reserved; set to binary zero.
070	002	Length of padding field, in bytes: xxx.
072	004	Reserved, set to binary zero.
076	016	Reserved, set to binary zero.
092	032	Reserved; set to binary zero.
124	Start of the optionally-encrypted secure subsection.	
124	008	Random number, confounder.
132	ppp	Prime number, p.
132 + ppp	qqq	Prime number, q
132 + ppp + qqq	rrr	$d_p = d \bmod(p - 1)$
132 + ppp + qqq + rrr	sss	$d_q = d \bmod(q - 1)$
132 + ppp + qqq + rrr + sss	uuu	$U = q^{-1} \bmod(p)$.
132 + ppp + qqq + rrr + sss + uuu	xxx	X'00' padding of length xxx bytes such that the length from the start of the random number above to the end of the padding field is a multiple of eight bytes.
	End of the optionally-encrypted secure subsection; all of the fields starting with the confounder field and ending with the variable length pad field are enciphered for key confidentiality when the key format-and-security flags (offset 28) indicate that the private key is enciphered. They are enciphered under a double-length transport key using the TDES (CBC outer chaining) algorithm.	
132 + ppp + qqq + rrr + sss + uuu + xxx	nnn	Modulus, n. n = pq where p and q are prime and $2^{512}<n<2^{2048}$.

Format of the DSS Private External Key Token

A DSS private external key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1E'
- A required DSS private key section, starting with the section identifier X'01'
- A required DSS public key section, starting with the section identifier X'03'
- An optional private key name section, starting with the section identifier X'10'

Table 105 presents the format of a DSS private external key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-justified and padded with zeros to the left.

Table 105. DSS Private External Key Token

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1E' indicates an external token. The private key is enciphered with a PKA master key.
001	001	Version, X'00'.
002	002	Length of the key token structure.
004	004	Ignored. Should be zero.
DSS Private Key Section and Secured Subsection (required)		
000	001	X'01', section identifier, DSS private key.
001	001	X'00', version.
002	002	Length of the DSS private key section, 436, X'01B4'.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key security: **X'00'** Unencrypted DSS private key subsection identifier. **X'81'** Encrypted DSS private key subsection identifier.
029	001	Padding, X'00'.
030	020	SHA-1 hash of the key token structure contents that follow the public key section. If no sections follow, this field is set to binary zeros.
050	010	Reserved; set to binary zero.
060	048	Ignored; set to binary zero.
108	128	Public key generator, g. $1<g<p$.
236	128	Prime modulus (large public modulus), p. $2^{L-1}<p<2^{L}$ and L (the modulus length) must be a multiple of 64.
364	020	Prime divisor (small public modulus), q. $2^{159}<q<2^{160}$.
384	004	Reserved; set to binary zero.
388	024	Random number, confounder. **Note:** This field and the next two fields are enciphered for key confidentiality when the key security flag (offset 28) indicates the private key is enciphered.
412	020	Secret DSS key, x; x is random. (See the preceding note.)

Table 105. DSS Private External Key Token (continued)

Offset (Dec)	Number of Bytes	Description
432	004	Random number, generated when the secret key is generated. (See the preceding note.)
DSS Public Key Section (required)		
000	001	X'03', section identifier, DSS public key.
001	001	X'00', version.
002	002	Section length, 14+yyy.
004	002	Size of p in bits. The size of p must be one of: 512, 576, 640, 704, 768, 832, 896, 960, or 1024.
006	002	Size of the p field in bytes, which is zero for a private token.
008	002	Size of the q field in bytes, which is zero for a private token.
010	002	Size of the g field in bytes, which is zero for a private token.
012	002	Size of the y field in bytes, “yyy”.
014	yyy	Public key, y. $y=g^x$ mod(p) **Note:** p, q, and y are defined in the DSS public key token.
Private Key Name (optional)		
000	001	X'10', section identifier, private key. name
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-justified, padded with space characters (X'20'). An access control system can use the private key name to verify that the calling application is entitled to use the key.

Format of the RSA Private Internal Key Token

An RSA private internal key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1F'
- basic record format of an RSA private internal key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-justified and padded with zeros to the left.

Table 106. RSA Private Internal Key Token Basic Record Format

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1F' indicates an internal token. The private key is enciphered with a PKA master key.
001	001	Version, X'00'.
002	002	Length of the key token structure excluding the internal information section.
004	004	Ignored; should be zero.

Table 106. RSA Private Internal Key Token Basic Record Format (continued)

Offset (Dec)	Number of Bytes	Description
RSA Private Key Section and Secured Subsection (required)		
• For 1024-bit X'02' Modulus-Exponent form refer to “RSA Private Key Token, 1024-bit Modulus-Exponent Internal Form for Cryptographic Coprocessor Feature” on page 376 • For 1024-bit X'06' Modulus-Exponent form refer to “RSA Private Key Token, 1024-bit Modulus-Exponent Internal Form for PCI Cryptographic Coprocessor” on page 376 • For 2048-bit X'08' Chinese Remainder Theorem form refer to “RSA Private Key Token, 2048-bit Chinese Remainder Theorem Internal Form” on page 378		
RSA Public Key Section (required)		
000	001	X'04', section identifier, RSA public key.
001	001	X'00', version.
002	002	Section length, 12+xxx.
004	002	Reserved field.
006	002	RSA public key exponent field length in bytes, “xxx”.
008	002	Public key modulus length in bits.
010	002	RSA public key modulus field length in bytes, which is zero for a private token.
012	xxx	Public key exponent (this is generally a 1, 3, or 64 to 256-byte quantity), e. e must be odd and 1<e<n. (Frequently, the value of e is $2^{16}+1$ (=65,537).
Private Key Name (optional)		
000	001	X'10', section identifier, private key name.
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-justified, padded with space characters (X'20'). An access control system can use the private key name to verify that the calling application is entitled to use the key.
Internal Information Section (required)		
000	004	Eye catcher 'PKTN'.
004	004	PKA token type. **Bit** **Meaning When Set On** **0** RSA key. **1** DSS key. **2** Private key. **3** Public key. **4** Private key name section exists. **5** Private key unenciphered. **6** Blinding information present. **7** Retained private key.
008	004	Address of token header.
012	002	Total length of total structure including this information section.
014	002	Count of number of sections.
016	016	PKA master key hash pattern.

Table 106. RSA Private Internal Key Token Basic Record Format (continued)

Offset (Dec)	Number of Bytes	Description
032	001	Domain of retained key.
033	008	Serial number of processor holding retained key.
041	007	Reserved.

RSA Private Key Token, 1024-bit Modulus-Exponent Internal Form for Cryptographic Coprocessor Feature

Table 107. RSA Private Internal Key Token, 1024-bit ME Form for Cryptographic Coprocessor Feature

Offset (Dec)	Number of Bytes	Description
000	001	X'02', section identifier, RSA private key.
001	001	X'00', version.
002	002	Length of the RSA private key section X'016C' (364 decimal).
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'02'** RSA private key.
029	001	Format of external key from which this token was derived: **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted.
030	020	SHA-1 hash of the key token structure contents that follow the public key section. If no sections follow, this field is set to binary zeros.
050	001	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zero.
051	009	Reserved; set to binary zero.
060	048	Object Protection Key (OPK) encrypted under a PKA master key—can be under the Signature Master Key (SMK) or Key Management Master Key (KMMK) depending on key use.
108	128	Secret key exponent d, encrypted under the OPK. $d=e^{-1} \bmod((p-1)(q-1))$
236	128	Modulus, n. n=pq where p and q are prime and $1<n<2^{1024}$.

RSA Private Key Token, 1024-bit Modulus-Exponent Internal Form for PCI Cryptographic Coprocessor

Table 108. RSA Private Internal Key Token, 1024-bit ME Form for PCI Cryptographic Coprocessor

Offset (Dec)	Number of Bytes	Description
000	001	X'06', section identifier, RSA private key modulus-exponent format (RSA-PRIV).
001	001	X'00', version.

Table 108. RSA Private Internal Key Token, 1024-bit ME Form for PCI Cryptographic Coprocessor (continued)

Offset (Dec)	Number of Bytes	Description
002	002	Length of the RSA private key section X'0198' (408 decimal) + rrr + iii + xxx.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to and including the modulus at offset 236.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'02'** RSA private key.
029	001	Format of external key from which this token was derived: **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted. **X'23'** Private key was generated using regeneration data. **X'24'** Private key was randomly generated.
030	020	SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, this field is set to binary zeros.
050	004	Key use flag bits. **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zeros.
054	006	Reserved; set to binary zero.
060	048	Object Protection Key (OPK) encrypted under the Asymmetric Keys Master Key using the ede3 algorithm.
108	128	Private key exponent d, encrypted under the OPK using the ede5 algorithm. $d=e^{-1}mod((p-1)(q-1))$, and $1<d<n$ where e is the public exponent.
236	128	Modulus, n. n=pq where p and q are prime and $2^{512}<n<2^{1024}$.
364	016	Asymmetric-Keys Master Key hash pattern.
380	020	SHA-1 hash value of the blinding information subsection cleartext, offset 400 to the end of the section.
400	002	Length of the random number r, in bytes: rrr.
402	002	Length of the random number r^{-1}, in bytes: iii.
404	002	Length of the padding field, in bytes: xxx.
406	002	Reserved; set to binary zeros.
408	Start of the encrypted blinding subsection	
408	rrr	Random number r (used in blinding).
408 + rrr	iii	Random number r^{-1} (used in blinding).
408 + rrr + iii	xxx	X'00' padding of length xxx bytes such that the length from the start of the encrypted blinding subsection to the end of the padding field is a multiple of eight bytes.
	End of the encrypted blinding subsection; all of the fields starting with the random number r and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) algorithm.	

RSA Private Key Token, 2048-bit Chinese Remainder Theorem Internal Form

This RSA private key token is supported on the PCI Cryptographic Coprocessor.

Table 109. RSA Private Internal Key Token, 2048-bit Chinese Remainder Theorem External Format

Offset (Dec)	Number of Bytes	Description
000	001	X'08', section identifier, RSA private key, CRT format (RSA-CRT)
001	001	X'00', version.
002	002	Length of the RSA private-key section, 132 + ppp + qqq + rrr + sss + uuu + +ttt + iii + xxx + nnn.
004	020	SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.
024	004	Reserved; set to binary zero.
028	001	Key format and security: **X'08'** Encrypted RSA private-key subsection identifier, Chinese Remainder form.
029	001	Key derivation method: **X'21'** External private key was specified in the clear. **X'22'** External private key was encrypted. **X'23'** Private key was generated using regeneration data. **X'24'** Private key was randomly generated.
030	020	SHA-1 hash of the optional key-name section and any following sections. If there are no optional sections, then 20 bytes of X'00'.
050	004	Key use flag bits: **Bit** **Meaning When Set On** **0** Key management usage permitted. **1** Signature usage not permitted. All other bits reserved, set to binary zero.
054	002	Length of prime number, p, in bytes: ppp.
056	002	Length of prime number, q, in bytes: qqq.
058	002	Length of d_p, in bytes: rrr.
060	002	Length of d_q, in bytes: sss.
062	002	Length of U, in bytes: uuu.
064	002	Length of modulus, n, in bytes: nnn.
066	002	Length of the random number r, in bytes: ttt.
068	002	Length of the random number r^{-1}, in bytes: iii.
070	002	Length of padding field, in bytes: xxx.
072	004	Reserved, set to binary zero.
076	016	Asymmetric-Keys Master Key hash pattern.
092	032	Object Protection Key (OPK) encrypted under the Asymmetric-Keys Master Key using the TDES (CBC outer chaining) algorithm.
124	Start of the encrypted secure subsection, encrypted under the OPK using TDES (CBC outer chaining).	
124	008	Random number, confounder.
132	ppp	Prime number, p.

Table 109. RSA Private Internal Key Token, 2048-bit Chinese Remainder Theorem External Format (continued)

Offset (Dec)	Number of Bytes	Description
132 + ppp	qqq	Prime number, q
132 + ppp + qqq	rrr	d_p = d mod(p - 1)
132 + ppp + qqq + rrr	sss	d_q = d mod(q - 1)
132 + ppp + qqq + rrr + sss	uuu	U = q^{-1}mod(p).
132 + ppp + qqq + rrr + sss + uuu	ttt	Random number r (used in blinding).
132 + ppp + qqq + rrr + sss + uuu + ttt	iii	Random number r^{-1} (used in blinding).
132 + ppp + qqq + rrr + sss + uuu + ttt + iii	xxx	X'00' padding of length xxx bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of eight bytes.
	End of the encrypted secure subsection; all of the fields starting with the confounder field and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) for key confidentiality.	
132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx	nnn	Modulus, n. n = pq where p and q are prime and 2^{512}<n<2^{2048}.

Format of the DSS Private Internal Key Token

A DSS private internal key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1F'
- A required DSS private key section, starting with the section identifier X'01'
- A required DSS public key section, starting with the section identifier X'03'
- An optional private key name section, starting with the section identifier X'10'
- A required internal information section, starting with the eyecatcher 'PKTN'

Table 110 presents the format of a DSS private internal token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-justified and padded with zeros to the left.

Table 110. DSS Private Internal Key Token

Offset (Dec)	Number of Bytes	Description
Token Header (required)		
000	001	Token identifier. X'1F' indicates an internal token. The private key is enciphered with a PKA master key.
001	001	Version, X'00'.
002	002	Length of the key token structure excluding the internal information section.
004	004	Ignored; should be zero.
DSS Private Key Section and Secured Subsection (required)		
000	001	X'01', section identifier, DSS private key.
001	001	X'00', version.

Table 110. DSS Private Internal Key Token (continued)

Offset (Dec)	Number of Bytes	Description
002	002	Length of the DSS private key section, 436, X'01B4'.
004	020	SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	004	Reserved; set to binary zero.
028	001	Key security: X'01' DSS private key.
029	001	Format of external key token: **X'10'** Private key generated on an ICSF host. **X'11'** External private key was specified in the clear. **X'12'** External private key was encrypted.
030	020	SHA-1 hash of the key token structure contents that follow the public key section. If no sections follow, this field is set to binary zeros.
050	010	Reserved; set to binary zero.
060	048	The OPK encrypted under a PKA master key (Signature Master Key (SMK)).
108	128	Public key generator, g. $1<g<p$.
236	128	Prime modulus (large public modulus), p. $2^{L-1}<p<2^L$ for $512 \le L \le 1024$, and L (the modulus length) must be a multiple of 64.
364	020	Prime divisor (small public modulus), q. $2^{159}<q<2^{160}$.
384	004	Reserved; set to binary zero.
388	024	Random number, confounder. **Note:** This field and the two that follow are enciphered under the OPK.
412	020	Secret DSS key, x. x is random. (See the preceding note.)
432	004	Random number, generated when the secret key is generated. (See the preceding note.)
DSS Public Key Section (required)		
000	001	X'03', section identifier, DSS public key.
001	001	X'00', version.
002	002	Section length, 14+yyy.
004	002	Size of p in bits. The size of p must be one of: 512, 576, 640, 704, 768, 832, 896, 960, or 1024.
006	002	Size of the p field in bytes, which is zero for a private token.
008	002	Size of the q field in bytes, which is zero for a private token.
010	002	Size of the g field in bytes, which is zero for a private token.
012	002	Size of the y field in bytes, "yyy".
014	yyy	Public key, y. $y=g^x \bmod(p)$; **Note:** p, g, and y are defined in the DSS public key token.
Private Key Name (optional)		
000	001	X'10', section identifier, private key name.
001	001	X'00', version.
002	002	Section length, X'0044' (68 decimal).
004	064	Private key name (in ASCII), left-justified, padded with space characters (X'20'). An access control system can use the private key name to verify that the calling application is entitled to use the key.

Table 110. DSS Private Internal Key Token (continued)

Offset (Dec)	Number of Bytes	Description
Internal Information Section (required)		
000	004	Eye catcher 'PKTN'.
004	004	PKA token type. **Bit** **Meaning When Set On** **0** RSA key. **1** DSS key. **2** Private key. **3** Public key. **4** Private key name section exists.
008	004	Address of token header.
012	002	Length of internal work area.
014	002	Count of number of sections.
016	016	PKA master key hash pattern.
032	016	Reserved.

PKA Null Key Token

Table 111 shows the format for a PKA null key token.

Table 111. Format of PKA Null Key Tokens

Bytes	Description
0	X'00' Token identifier (indicates that this is a null key token).
1	Version, X'00'
2–3	X'0008' Length of the key token structure.
4–7	Ignored (should be zero).

Appendix C. Control Vectors and Changing Control Vectors with the CVT Callable Service

This section contains a control vector table which displays the default value of the control vector that is associated with each type of key. It also describes how to change control vectors with the control vector translate callable service.

Control Vector Table

Note: The Control Vectors used in ICSF are exactly the same as documented in CCA and the TSS manuals.

The master key enciphers all keys operational on your system. A transport key enciphers keys that are distributed off your system. Before a master key or transport key enciphers a key, ICSF exclusive ORs both halves of the master key or transport key with a control vector. The same control vector is exclusive ORed to the left and right half of a master key or transport key.

Also, if you are entering a key part, ICSF exclusive ORs each half of the key part with a control vector before placing the key part into the CKDS.

Each type of key on ICSF (except the master key) has either one or two unique control vectors associated with it. The control vector that ICSF exclusive ORs the master key or transport key with depends on the type of key the master key or transport key is enciphering. For double-length keys, a unique control vector exists for each half of a specific key type. For example, there is a control vector for the left half of an input PIN-encrypting key, and a control vector for the right half of an input PIN-encrypting key.

If you are entering a key part into the CKDS, ICSF exclusive ORs the key part with the unique control vector(s) associated with the key type. ICSF also enciphers the key part with two master key variants for a key part. One master key variant enciphers the left half of the key part, and another master key variant enciphers the right half of the key part. ICSF creates the master key variants for a key part by exclusive ORing the master key with the control vectors for key parts. These procedures protect key separation.

Table 112 displays the default value of the control vector that is associated with each type of key. Some key types do not have a default control vector. For keys that are double-length, ICSF enciphers a unique control vector on each half. Control vectors indicated with an ″*″ are supported by the Cryptographic Coprocessor Feature.

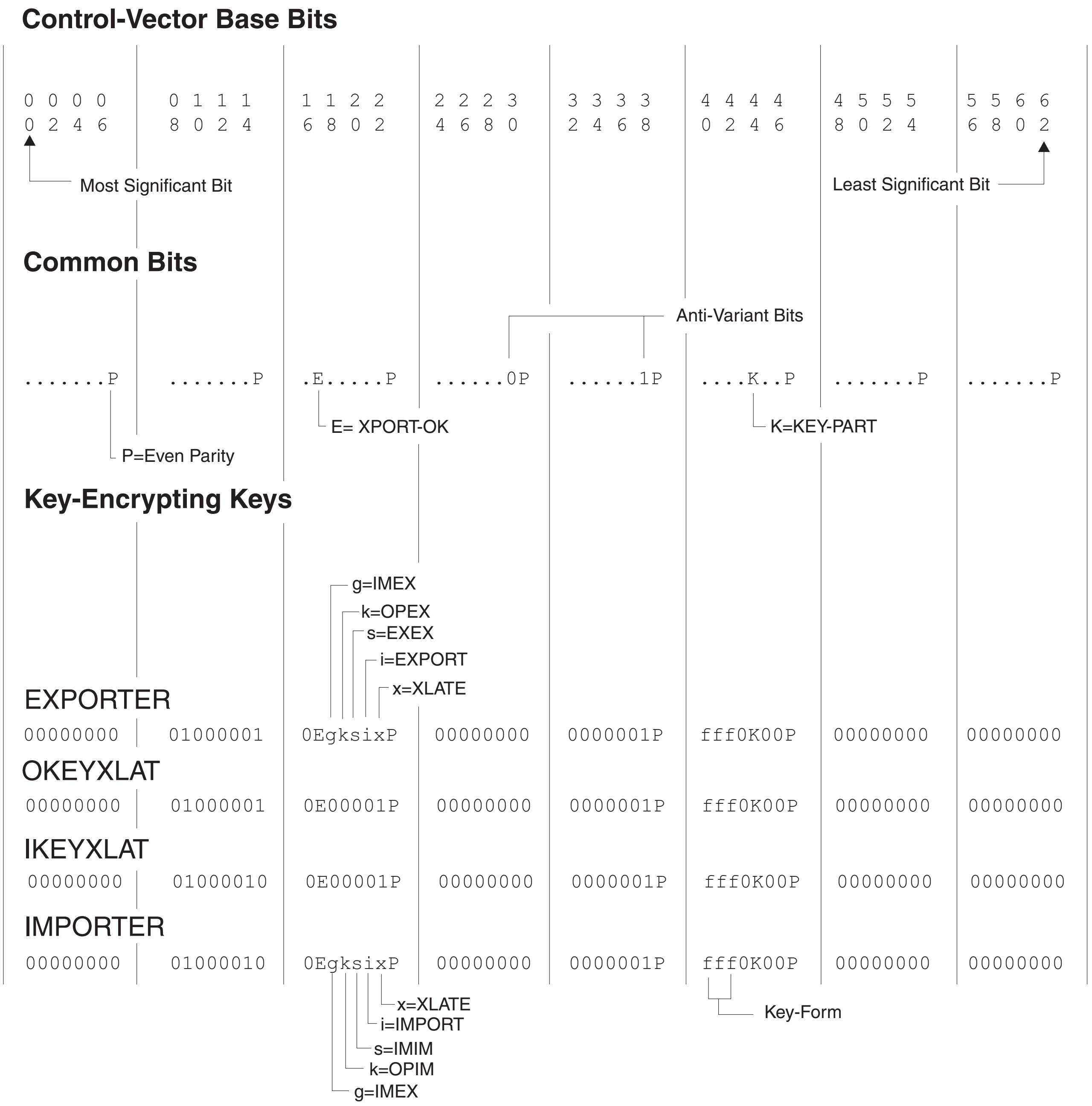
Table 112. Default Control Vector Values

Key Type	Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key	Control Vector Value (Hex) Value for Right Half of Double-length Key
*AKEK	00 00 00 00 00 00 00 00	
CIPHER	00 03 71 00 03 00 00 00	
CVARDEC	00 3F 42 00 03 00 00 00	
CVARENC	00 3F 48 00 03 00 00 00	

Table 112. Default Control Vector Values (continued)

Key Type	**Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key**	**Control Vector Value (Hex) Value for Right Half of Double-length Key**
CVARPINE	00 3F 41 00 03 00 00 00	
CVARXCVL	00 3F 44 00 03 00 00 00	
CVARXCVR	00 3F 47 00 03 00 00 00	
*DATA	00 00 00 00 00 00 00 00	
DATAC	00 00 71 00 03 41 00 00	00 00 71 00 03 21 00 00
*DATAM generation key (external)	00 00 4D 00 03 41 00 00	00 00 4D 00 03 21 00 00
*DATAM key (internal)	00 05 4D 00 03 00 00 00	00 05 4D 00 03 00 00 00
*DATAMV MAC verification key (external)	00 00 44 00 03 41 00 00	00 00 44 00 03 21 00 00
*DATAMV MAC verification key (internal)	00 05 44 00 03 00 00 00	00 05 44 00 03 00 00 00
*DATAXLAT	00 06 71 00 03 00 00 00	
DECIPHER	00 03 50 00 03 00 00 00	
DKYGENKY	00 71 44 00 03 41 00 00	00 71 44 00 03 21 00 00
ENCIPHER	00 03 60 00 03 00 00 00	
*EXPORTER	00 41 7D 00 03 41 00 00	00 41 7D 00 03 21 00 00
IKEYXLAT	00 42 42 00 03 41 00 00	00 42 42 00 03 21 00 00
*IMP-PKA	00 42 05 00 03 41 00 00	00 42 05 00 03 21 00 00
*IMPORTER	00 42 7D 00 03 41 00 00	00 42 7D 00 03 21 00 00
*IPINENC	00 21 5F 00 03 41 00 00	00 21 5F 00 03 21 00 00
*MAC	00 05 4D 00 03 00 00 00	
*MACVER	00 05 44 00 03 00 00 00	
OKEYXLAT	00 41 42 00 03 41 00 00	00 41 42 00 03 21 00 00
*OPINENC	00 24 77 00 03 41 00 00	00 24 77 00 03 21 00 00
*PINGEN	00 22 7E 00 03 41 00 00	00 22 7E 00 03 21 00 00
*PINVER	00 22 42 00 03 41 00 00	00 22 42 00 03 21 00 00

Note: The external control vectors for DATAC, DATAM MAC generation and DATAMV MAC verification keys are also referred to as data compatibility control vectors.



	0 0 0 0 0 2 4 6	0 1 1 1 8 0 2 4	1 1 2 2 6 8 0 2	2 2 2 3 4 6 8 0	3 3 3 3 2 4 6 8	4 4 4 4 0 2 4 6	4 5 5 5 8 0 2 4	5 5 6 6 6 8 0 2
Common Bits	.......P	.......P	.E.....P	......0P	......1P	....K..P	.......P	.......P
EXPORTER	00000000	01000001	0EgksixP	00000000	0000001P	fff0K00P	00000000	00000000
OKEYXLAT	00000000	01000001	0E00001P	00000000	0000001P	fff0K00P	00000000	00000000
IKEYXLAT	00000000	01000010	0E00001P	00000000	0000001P	fff0K00P	00000000	00000000
IMPORTER	00000000	01000010	0EgksixP	00000000	0000001P	fff0K00P	00000000	00000000

Figure 3. Control Vector Base Bit Map (Common Bits and Key-Encrypting Keys)

Control-Vector Base Bits

0 0 0 0	0 1 1 1	1 1 2 2	2 2 2 3	3 3 3 3	4 4 4 4	4 5 5 5	5 5 6 6
0 2 4 6	8 0 2 4	6 8 0 2	4 6 8 0	2 4 6 8	0 2 4 6	8 0 2 4	6 8 0 2

Most Significant Bit

Least Significant Bit

Data Operation Keys

DATA							
00000000	00000000	0Eedmv0P	00000000	00000011	fff0K00P	00000000	00000000
DATAC							
00000000	00000000	0E11000P	00000000	00000011	fff0K00P	00000000	00000000
DATAM							
00000000	00000000	0E00110P	00000000	00000011	fff0K00P	00000000	00000000
DATAMV							
00000000	00000000	0E00010P	00000000	00000011	fff0K00P	00000000	00000000
CIPHER							
00000000	00000011	0E11000P	00000000	00000011	fff0K00P	00000000	00000000
DECIPHER							
00000000	00000011	0E01000P	00000000	00000011	fff0K00P	00000000	00000000
ENCIPHER							
00000000	00000011	0E10000P	00000000	00000011	fff0K00P	00000000	00000000
SECMSG							
00000000	00001010	0E..000P	00000000	00000011	fff0K00P	00000000	00000000
		01 PIN encryption 10 Key encryption					
MAC							
cccc0000	00000101	0E00110P	00000000	00000011	fff0K00P	00000000	00000000
MACVER							
cccc0000	00000101	0E00010P	00000000	00000011	fff0K00P	00000000	00000000

0000 ANY
0001 ANSI X9.9
0010 CVV KEY-A
0011 CVV KEY-B

Key-Form

Figure 4. Control Vector Base Bit Map (Data Operation Keys)

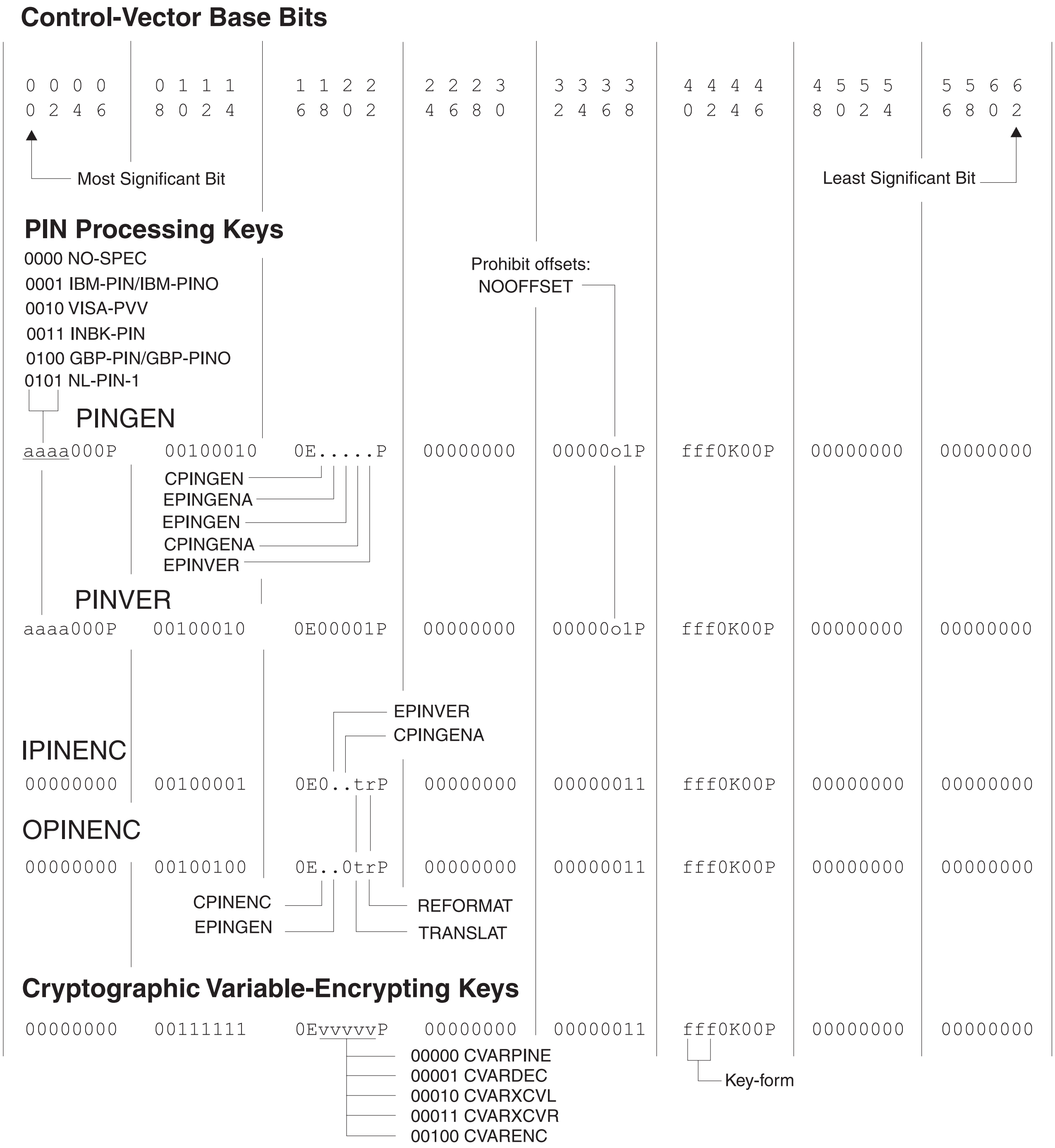


Figure 5. Control Vector Base Bit Map (PIN Processing Keys and Cryptographic Variable-Encrypting Keys)

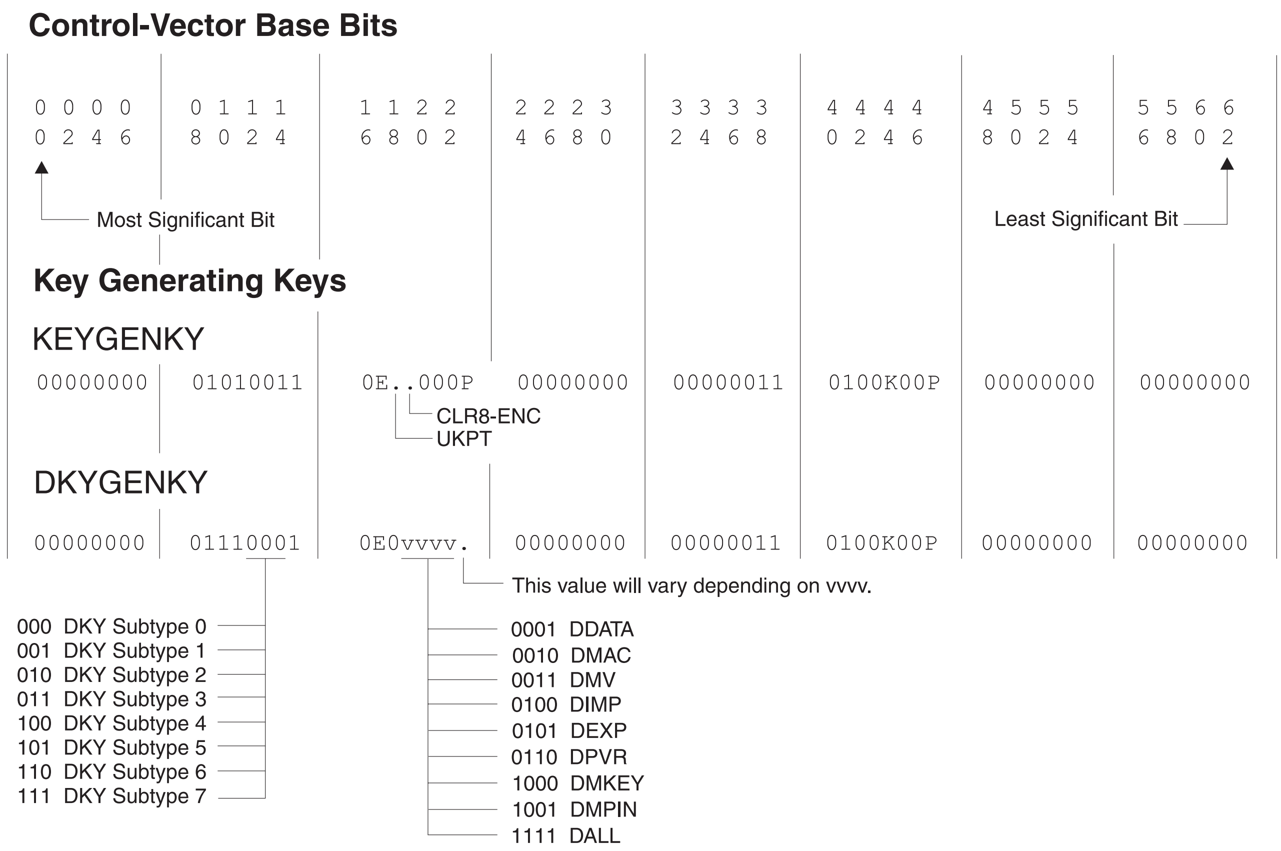


Figure 6. Control Vector Base Bit Map (Key Generating Keys)

Key Form Bits, ’fff’ - The key form bits, 40-42, and for a double-length key, bits 104-106, are designated ’fff’ in the preceding illustration. These bits can have these values:

000	Single length key
010	Double length key, left half
001	Double length key. right half

The following values may exist in some CCA implementations:

110	Double-length key, left half, halves guaranteed unique
101	Double-length key, right half, halves guaranteed unique

Specifying a Control-Vector-Base Value

You can determine the value of a control vector by working through the following series of questions:

1. Begin with a field of 64 bits (eight bytes) set to B'0'. The most significant bit is referred to as bit 0. Define the key type and subtype (bits 8 to 14), as follows:
   - The main key type bits (bits 8 to 11). Set bits 8 to 11 to one of the following values:

Bits 8 to 11	Main Key Type
0000	Data operation keys
0010	PIN keys
0011	Cryptographic variable-encrypting keys
0100	Key-encrypting keys
0101	Key-generating keys
0111	Diversified key-generating keys

- The key subtype bits (bits 12 to 14). Set bits 12 to 14 to one of the following values:

  Note: For Diversified Key Generating Keys, the subtype field specifies the hierarchical level of the DKYGENKY. If the subtype is non-zero, then the DKYGENKY can only generate another DKYGENKY key with the hierarchy level decremented by one. If the subtype is zero, the DKYGENKY can only generate the final diversified key (a non-DKYGENKY key) with the key type specified by the usage bits.

Bits 12 to 14	Key Subtype
Data Operation Keys	
000	Compatibility key (DATA)
001	Confidentiality key (CIPHER, DECIPHER, or ENCIPHER)
010	MAC key (MAC or MACVER)
101	Secure messaging keys
Key-Encrypting Keys	
000	Transport-sending keys (EXPORTER and OKEYXLAT)
001	Transport-receiving keys (IMPORTER and IKEYXLAT)
PIN Keys	
001	PIN-generating key (PINGEN, PINVER)
000	Inbound PIN-block decrypting key (IPINENC)
010	Outbound PIN-block encrypting key (OPINENC)
Cryptographic Variable-Encrypting Keys	
111	Cryptographic variable-encrypting key (CVAR....)
Diversified Key Generating Keys	
000	DKY Subtype 0
001	DKY Subtype 1
010	DKY Subtype 2
011	DKY Subtype 3
100	DKY Subtype 4
101	DKY Subtype 5
110	DKY Subtype 6
111	DKY Subtype 7

2. For key-encrypting keys, set the following bits:
   - The key-generating usage bits (gks, bits 18 to 20). Set the gks bits to B'111' to indicate that the Key Generate callable service can use the associated

key-encrypting key to encipher generated keys when the Key Generate callable service is generating various key-pair key-form combinations (see the Key-Encrypting Keys section of Figure 3). Without any of the gks bits set to 1, the Key Generate callable service cannot use the associated key-encrypting key. The Key Token Build callable service can set the gks bits to 1 when you supply the **OPIM, IMEX, IMIM, OPEX**, and **EXEX** keywords.

- The IMPORT and EXPORT bit and the XLATE bit (ix, bits 21 and 22). If the ‘i’ bit is set to 1, the associated key-encrypting key can be used in the Data Key Import, Key Import, Data Key Export, and Key Export callable services. If the ‘x’ bit is set to 1, the associated key-encrypting key can be used in the Key Translate callable service.
- The key-form bits (fff, bits 40 to 42). The key-form bits indicate how the key was generated and how the control vector participates in multiple-enciphering. To indicate that the parts can be the same value, set these bits to B'010'. For information about the value of the key-form bits in the right half of a control vector, see Step 8.

3. For MAC and MACVER keys, set the following bits:
   - The MAC control bits (bits 20 and 21). For a MAC-generate key, set bits 20 and 21 to B'11'. For a MAC-verify key, set bits 20 and 21 to B'01'.
   - The key-form bits (fff, bits 40 to 42). For a single-length key, set the bits to B'000'. For a double-length key, set the bits to B'010'.
4. For PINGEN and PINVER keys, set the following bits:
   - The PIN calculation method bits (aaaa, bits 0 to 3). Set these bits to one of the following values:

Bits 0 to 3	Calculation Method Keyword	Description
0000	NO-SPEC	A key with this control vector can be used with any PIN calculation method.
0001	IBM-PIN or IBM-PINO	A key with this control vector can be used only with the IBM PIN or PIN Offset calculation method.
0010	VISA-PVV	A key with this control vector can be used only with the VISA-PVV calculation method.
0100	GBP-PIN or GBP-PINO	A key with this control vector can be used only with the German Banking Pool PIN or PIN Offset calculation method.
0011	INBK-PIN	A key with this control vector can be used only with the Interbank PIN calculation method.
0101	NL-PIN-1	A key with this control vector can be used only with the NL-PIN-1, Netherlands PIN calculation method.

   - The prohibit-offset bit (o, bit 37) to restrict operations to the PIN value. If set to 1, this bit prevents operation with the IBM 3624 PIN Offset calculation method and the IBM German Bank Pool PIN Offset calculation method.

5. For PINGEN, IPINENC, and OPINENC keys, set bits 18 to 22 to indicate whether the key can be used with the following callable services

Service Allowed	Bit Name	Bit
Clear PIN Generate	CPINGEN	18
Encrypted PIN Generate Alternate	EPINGENA	19
Encrypted PIN Generate	EPINGEN	20 for PINGEN 19 for OPINENC
Clear PIN Generate Alternate	CPINGENA	21 for PINGEN 20 for IPINENC
Encrypted Pin Verify	EPINVER	19
Clear PIN Encrypt	CPINENC	18

6. For the IPINENC (inbound) and OPINENC (outbound) PIN-block ciphering keys, do the following:
   - Set the TRANSLAT bit (t, bit 21) to 1 to permit the key to be used in the PIN Translate callable service. The Control Vector Generate callable service can set the TRANSLAT bit to 1 when you supply the **TRANSLAT** keyword.
   - Set the REFORMAT bit (r, bit 22) to 1 to permit the key to be used in the PIN Translate callable service. The Control Vector Generate callable service can set the REFORMAT bit and the TRANSLAT bit to 1 when you supply the **REFORMAT** keyword.
7. For the cryptographic variable-encrypting keys (bits 18 to 22), set the variable-type bits (bits 18 to 22) to one of the following values:

Bits 18 to 22	Generic Key Type	Description
00000	CVARPINE	Used in the Encrypted PIN Generate Alternate service to encrypt a clear PIN.
00010	CVARXCVL	Used in the Control Vector Translate callable service to decrypt the left mask array.
00011	CVARXCVR	Used in the Control Vector Translate callable service to decrypt the right mask array.
00100	CVARENC	Used in the Cryptographic Variable Encipher callable service to encrypt an unformatted PIN.

8. For key-generating keys, set the following bits:
   - For KEYGENKY, set bit 18 for UKPT usage and bit 19 for CLR8-ENC usage.
   - For DKYGENKY, bit 18 is reserved and must be zero.
   - Set bit 19 to 1 if the key will be used in the diverisifed key generate service to generate a diversified key. Usage bits 18-22 for the DKYGENKY key type are defined as follows. They will be encoded as the final key type that the DKYGENKY key generates.

Bits 19 to 22	Keyword	Usage
0001	DDATA	DATA, DATAC, single or double length
0010	DMAC	MAC, DATAM
0011	DMV	MACVER, DATAMV
0100	DIMP	IMPORTER, IKEYXLAT
0101	DEXP	EXPORTER, OKEYXLAT
0110	DPVR	PINVER
1000	DMKEY	Secure message key for encrypting keys
1001	DMPIN	Secure message key for encrypting PINs
1111	DALL	All key types may be generated except DKYGENKY and KEYGENKY keys. Usage of the DALL keyword is controlled by a separate access control point.

9. For secure messaging keys, set the following bits:
   - Set bit 18 to 1 if the key will be used in the secure messaging for PINs service. Set bit 19 to 1 if the key will be used in the secure messaging for keys service.
10. For all keys, set the following bits:
    - The export bit (E, bit 17). If set to 0, the export bit prevents a key from being exported. By setting this bit to 0, you can prevent the receiver of a key from exporting or translating the key for use in another cryptographic subsystem. Once this bit is set to 0, it cannot be set to 1 by any service other than Control Vector Translate. The Prohibit Export callable service can reset the export bit.
    - The key-part bit (K, bit 44). Set the key-part bit to 1 in a control vector associated with a key part. When the final key part is combined with previously accumulated key parts, the key-part bit in the control vector for the final key part is set to 0. The Control Vector Generate callable service can set the key-part bit to 1 when you supply the **KEY-PART** keyword.
    - The anti-variant bits (bit 30 and bit 38). Set bit 30 to 0 and bit 38 to 1. Many cryptographic systems have implemented a system of variants where a 7-bit value is exclusive-ORed with each 7-bit group of a key-encrypting key before enciphering the target key. By setting bits 30 and 38 to opposite values, control vectors do not produce patterns that can occur in variant-based systems.
    - Control vector bits 64 to 127. If bits 40 to 42 are B'000' (single-length key), set bits 64 to 127 to 0. Otherwise, copy bits 0 to 63 into bits 64 to 127 and set bits 105 and 106 to B'01'.
    - Set the parity bits (low-order bit of each byte, bits 7, 15, ..., 127). These bits contain the parity bits (P) of the control vector. Set the parity bit of each byte so the number of zero-value bits in the byte is an even number.
    - For secure messaging keys, usage bit 18 on will enable the encryption of keys in a secure message and usage bit 19 on will enable the encryption of PINs in a secure message.

Changing Control Vectors with the Control Vector Translate Callable Service

Do the following when using the Control Vector Translate callable service:

- Provide the control information for testing the control vectors of the source, target, and key-encrypting keys to ensure that only sanctioned changes can be performed
- Select the key-half processing mode.

Providing the Control Information for Testing the Control Vectors

To minimize your security exposure, the Control Vector Translate callable service requires control information (*mask array* information) to limit the range of allowable control vector changes. To ensure that this service is used only for authorized purposes, the source-key control vector, target-key control vector, and key-encrypting key (KEK) control vector must pass specific tests. The tests on the control vectors are performed within the secured cryptographic engine.

The tests consist of evaluating four logic expressions, the results of which must be a string of binary zeros. The expressions operate bitwise on information that is contained in the mask arrays and in the portions of the control vectors associated with the key or key-half that is being processed. If any of the expression evaluations do not result in all zero bits, the callable service is ended with a *control vector violation* return and reason code (8/39). See Figure 7. Only the 56 bit positions that are associated with a key value are evaluated. The low-order bit that is associated with key parity in each key byte is not evaluated.

Mask Array Preparation

A mask array consists of seven 8-byte elements: A_1, B_1, A_2, B_2, A_3, B_3, and B_4. You choose the values of the array elements such that each of the following four expressions evaluates to a string of binary zeros. (See Figure 7 on page 395.) Set the **A** bits to the value that you require for the corresponding control vector bits. In expressions 1 through 3, set the **B** bits to select the control vector bits to be evaluated. In expression 4, set the **B** bits to select the source and target control vector bits to be evaluated. Also, use the following control vector information:

C_1 is the control vector associated with the left half of the KEK.

C_2 is the control vector associated with the source key, or selected source-key half/halves.

C_3 is the control vector associated with the target key or selected target-key half/halves.

1. (C_1 exclusive-OR A_1) logical-AND B_1

   This expression tests whether the KEK used to encipher the key meets your criteria for the desired translation.
2. (C_2 exclusive-OR A_2) logical-AND B_2

   This expression tests whether the control vector associated with the source key meets your criteria for the desired translation.
3. (C_3 exclusive-OR A_3) logical-AND B_3

   This expression tests whether the control vector associated with the target key meets your criteria for the desired translation.
4. (C_2 exclusive-OR C_3) logical-AND B_4

   This expression tests whether the control vectors associated with the source key and the target key meet your criteria for the desired translation.

Encipher two copies of the mask array, each under a different cryptographic-variable key (key type CVARENC). To encipher each copy of the mask array, use the Cryptographic Variable Encipher callable service. Use two different keys so that the enciphered-array copies are unique values. When using the Control Vector Translate callable service, the *mask_array_left* parameter and the *mask_array_right* parameter identify the enciphered mask arrays. The *array_key_left* parameter and the *array_key_right* parameter identify the internal keys for deciphering the mask arrays. The *array_key_left* key must have a key type of CVARXCVL and the array_key_right key must have a key type of CVARXCVR. The cryptographic process deciphers the arrays and compares the results; for the service to continue, the deciphered arrays must be equal. If the results are not equal, the service returns the return and reason code for data that is not valid (8/385).

Use the Key Generate callable service to create the key pairs CVARENC-CVARXCVL and CVARENC-CVARXCVR. Each key in the key pair must be generated for a different node. The CVARENC keys are generated for, or imported into, the node where the mask array will be enciphered. After enciphering the mask array, you should destroy the enciphering key. The CVARXCVL and CVARXCVR keys are generated for, or imported into, the node where the Control Vector Translate callable service will be performed.

If using the **BOTH** keyword to process both halves of a double-length key, remember that bits 41, 42, 104, and 105 are different in the left and right halves of the CCA control vector and must be ignored in your mask-array tests (that is, make the corresponding $\mathbf{B_2}$ and/or $\mathbf{B_3}$ bits equal to zero).

When the control vectors pass the masking tests, the verb does the following:

- Deciphers the source key. In the decipher process, the service uses a key that is formed by the exclusive-OR of the KEK and the control vector in the key token variable the *source_key_token* parameter identifies.
- Enciphers the deciphered source key. In the encipher process, the service uses a key that is formed by the exclusive-OR of the KEK and the control vector in the key token variable the *target_key_token* parameter identifies.
- Places the enciphered key in the key field in the key token variable the *target_key_token* parameter identifies.

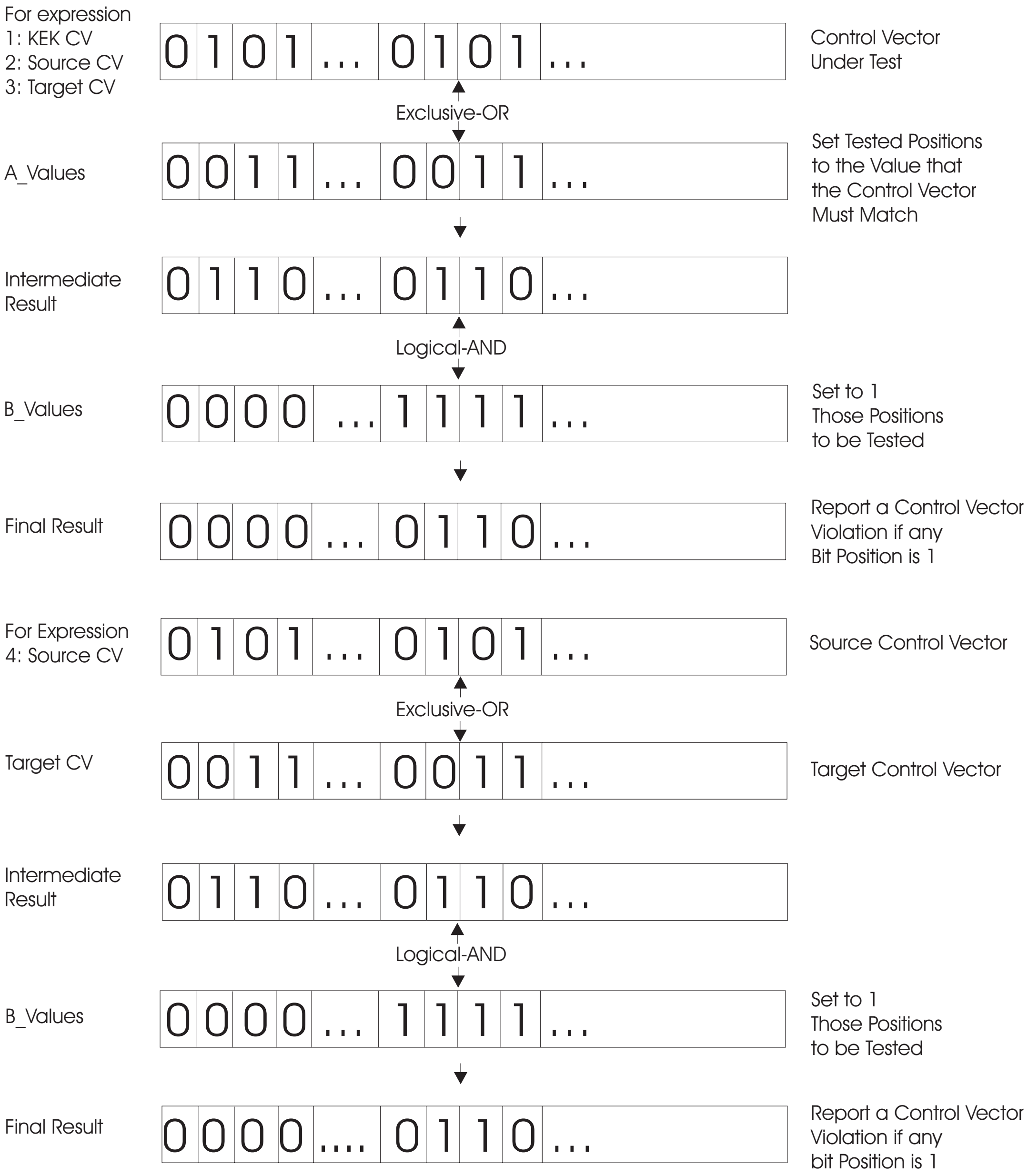


Figure 7. Control Vector Translate Callable Service Mask_Array Processing

Selecting the Key-Half Processing Mode

Use the Control Vector Translate callable service to change a control vector associated with a key. Rule-array keywords determine which key halves are processed in the call, as shown in Figure 8 on page 396.

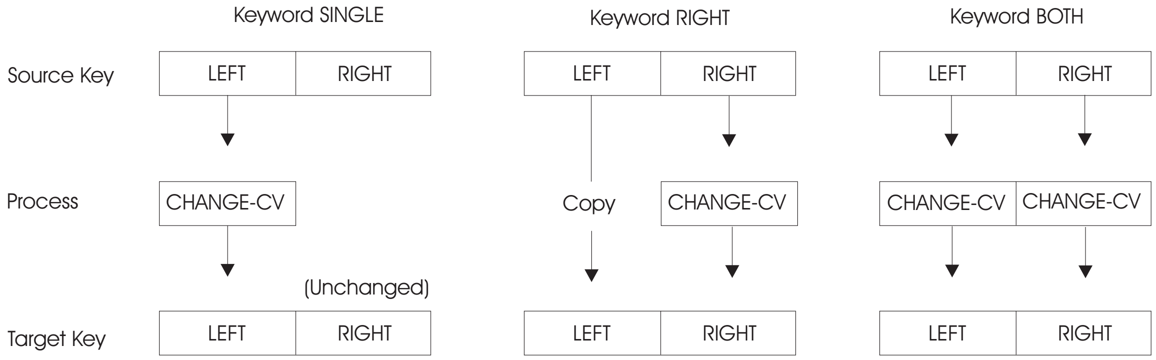


Figure 8. Control Vector Translate Callable Service. In this figure, CHANGE-CV means the requested control vector translation change; LEFT and RIGHT mean the left and right halves of a key and its control vector.

Keyword	Meaning

SINGLE
This keyword causes the control vector of the left half of the source key to be changed. The updated key half is placed into the left half of the target key in the target key token. The right half of the target key is unchanged.

The **SINGLE** keyword is useful when processing a single-length key, or when first processing the left half of a double-length key (to be followed by processing the right half).

RIGHT
This keyword causes the control vector of the right half of the source key to be changed. The updated key half is placed into the right half of the target key of the target key token. The left half of the source key is copied unchanged into the left half of the target key in the target key token.

BOTH
This keyword causes the control vector of both halves of the source key to be changed. The updated key is placed into the target key in the target key token.

A single set of control information must permit the control vector changes applied to each key half. Normally, control vector bit positions 41, 42, 105, and 106 are different for each key half. Therefore, set bits 41 and 42 to B'00' in mask array elements B_1, B_2, and B_3.

You can verify that the source and target key tokens have control vectors with matching bits in bit positions 40-42 and 104-106, the “form field” bits. Ensure that bits 40-42 of mask array B_4 are set to B'111'.

LEFT
This keyword enables you to supply a single-length key and obtain a double-length key. The source key token must contain:

- The KEK-enciphered single-length key
- The control vector for the single-length key (often this is a null value)
- A control vector, stored in the source token where the right-half control vector is normally stored, used in decrypting the single-length source key when the key is being processed for the target right half of the key.

The service first processes the source and target tokens as with the **SINGLE** keyword. Then the source token is processed using the single-length enciphered key and the source token right-half control vector to obtain the actual key value. The key value is then enciphered using the KEK and the control vector in the target token for the right-half of the key.

This approach is frequently of use when you must obtain a double-length CCA key from a system that only supports a single-length key, for example when processing PIN keys or key-encrypting keys received from non-CCA systems.

To prevent the service from ensuring that each key byte has odd parity, you can specify the **NOADJUST** keyword. If you do not specify the **NOADJUST** keyword, or if you specify the **ADJUST** keyword, the service ensures that each byte of the target key has odd parity.

When the Target Key-Token CV Is Null

When you use any of the **LEFT**, **BOTH**, or **RIGHT** keywords, and when the control vector in the target key token is null (all B'0'), then bit 0 in byte 59 will be set to B'1' to indicate that this is a double-length DATA key.

Control Vector Translate Example

As an example, consider the case of receiving a single-length PIN-block encrypting key from a non-CCA system. Often such a key will be encrypted by an unmodified transport key (no control vector or variant is used). In a CCA system, an inbound PIN encrypting key is double-length.

First use the Key Token Build callable service to insert the single-length key value into the left-half key-space in a key token. Specify **USE-CV** as a key type and a control vector value set to 16 bytes of X'00'. Also specify **EXTERNAL**, **KEY**, and **CV** keywords in the rule array. This key token will be the source key key-token.

Second, the target key token can also be created using the Key Token Build callable service. Specify a key type of **IPINENC** and the **NO-EXPORT** rule array keyword.

Then call the Control Vector Translate callable service and specify a rule-array keyword of **LEFT**. The mask arrays can be constructed as follows:

- A_1 is set to the value of the KEK's control vector, most likely the value of an IMPORTER key, perhaps with the NO-EXPORT bit set. B_1 is set to eight bytes of X'FF' so that all bits of the KEK's control vector will be tested.
- A_2 is set to eight bytes of X'00', the (null) value of the source key control vector. B_2 is set to eight bytes of X'FF' so that all bits of the source-key "control vector" will be tested.
- A_3 is set to the value of the target key's left-half control vector. B_3 is set to X'FFFF FFFF FF9F FFFF'. This will cause all bits of the control vector to be tested except for the two ("fff") bits used to distinguish between the left-half and right-half target-key control vector.
- B_4 is set to eight bytes of X'00' so that no comparison is made between the source and target control vectors.

Appendix D. Coding Examples

This appendix provides sample routines using the ICSF callable services for the following languages:

- C
- COBOL
- Assembler
- PL/1

The COBOL and Assembler H examples that follow use the key generate, encipher, and decipher callable services to determine whether the deciphered text matches the starting text.

C

```
/*-------------------------------------------------------------------*
 * Example using C:                                                  *
 *   Invokes CSNBKGN (key generate), CSNBENC (DES encipher) and      *
 *   CSNBDEC (DES decipher)                                          *
 *-------------------------------------------------------------------*/
#include <stdio.h>
#include "csfhdrs.h"

/*-------------------------------------------------------------------*
 * Prototypes for functions in this example                          *
 *-------------------------------------------------------------------*/

/*-------------------------------------------------------------------*
 * Utility for printing hex strings                                  *
 *-------------------------------------------------------------------*/
void printHex(unsigned char *, unsigned int);

/*********************************************************************/
/* Main Function                                                     */
/*********************************************************************/
int main(void) {

  /*-----------------------------------------------------------------*
   * Constant inputs to ICSF services                                *
   *-----------------------------------------------------------------*/
  static int textLen = 24;
  static unsigned char clearText[24]="ABCDEFGHIJKLMN0987654321";
  static unsigned char cipherProcessRule[8]="CUSP    ";
  static unsigned char keyForm[4]="OP  ";
  static unsigned char keyLength[8]="SINGLE  ";
  static unsigned char dataKeyType[8]="DATA    ";
  static unsigned char nullKeyType[8]="        ";
  static unsigned char ICV[8]={0};
  static unsigned char pad[1]={0};
  static int exitDataLength = 0;
  static unsigned char exitData[4]={0};
  static int ruleArrayCount = 1;

  /*-----------------------------------------------------------------*
   * Variable inputs/outputs for ICSF services           *
   *-----------------------------------------------------------------*/
  unsigned char cipherText[24]={0};
  unsigned char compareText[24]={0};
  unsigned char dataKeyId[64]={0};
  unsigned char nullKeyId[64]={0};
  unsigned char dummyKEKKeyId1[64]={0};
  unsigned char dummyKEKKeyId2[64]={0};
  int returnCode = 0;
  int reasonCode = 0;
  unsigned char OCV[18]={0};

  /*-----------------------------------------------------------------*
   * Begin executable code                                           *
   *-----------------------------------------------------------------*/
  do {
    /*---------------------------------------------------------------*
     * Call key generate                                             *
     *---------------------------------------------------------------*/
    if ((returnCode = CSNBKGN(&returnCode,
                              &reasonCode,
                              &exitDataLength,
                              exitData,
                              keyForm,
                              &keyLength,
                              dataKeyType,
                              nullKeyType,
                              dummyKEKKeyId1,
                              dummyKEKKeyId2,
                              dataKeyId,
                              nullKeyId)) != 0) {
      printf("\nKey Generate failed:\;n");
      printf("   Return Code = %04d\n",returnCode);
      printf("   Reason Code = %04d\n",reasonCode);
      break;
      }
   /*---------------------------------------------------------------*
    * Call encipher                                                 *
    *---------------------------------------------------------------*/
   printf("\nClear Text\n");
   printHex(clearText,sizeof(clearText));

   if ((returnCode = CSNBENC(&returnCode,
                             &reasonCode,
                             &exitDataLength,
                             exitData,
                             dataKeyId,
                             &textLen,
                             clearText,
                             ICV,
                             &ruleArrayCount,
                             cipherProcessRule,
                             pad,
                             OCV,
                             cipherText)) != 0) {
     printf("\nReturn from Encipher:\n");
     printf("   Return Code = %04d\n",returnCode);
     printf("   Reason Code = %04d\n",reasonCode);
     if (returnCode > 4)
       break;
     }

   /*---------------------------------------------------------------*
    * Call decipher                                                 *
    *---------------------------------------------------------------*/
   printf("\nCipher Text\n");
   printHex(cipherText,sizeof(cipherText));

   if ((returnCode = CSNBDEC(&returnCode,
                             &reasonCode,
                             &exitDataLength,
                             exitData,
                             dataKeyId,
                             &textLen,
                             cipherText,
                             ICV,
                             &ruleArrayCount,
                             cipherProcessRule,
                             OCV,
                             compareText)) != 0) {
     printf("\nReturn from Decipher:\n");
     printf("   Return Code = %04d\n",returnCode);
     printf("   Reason Code = %04d\n",reasonCode);
     if (returnCode > 4)
       break;
     }

   /*---------------------------------------------------------------*
    * End                                                           *
    *---------------------------------------------------------------*/
   printf("\nClear Text after decipher\n");
   printHex(compareText,sizeof(compareText));

   } while(0);

   return returnCode;

} /* end main */

void printHex (unsigned char * text, unsigned int len)
/*------------------------------------------------------------------*
 * Prints a string as hex characters                                *
 *------------------------------------------------------------------*/

{
  unsigned int i;

  for (i = 0; i < len; ++i)
    if ( ((i & 7) == 7) || (i == (len - 1)) )
      printf ("  %02x\n", text&[i]);
    else
      printf ("  %02x", text[i]);
  printf ("\n");
} /* end printHex */
```

COBOL

```
**************************
 IDENTIFICATION DIVISION.
***************************
 PROGRAM-ID. COBOLXMP.
*****************************
 ENVIRONMENT DIVISION.
*****************************************************************
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  IBM-370.
 OBJECT-COMPUTER.  IBM-370.
*****************************
 DATA DIVISION.
*****************************************************************
 FILE SECTION.
 WORKING-STORAGE SECTION.
 77  INPUT-TEXT                   PIC       X(24)
     VALUE 'ABCDEFGHIJKLMN0987654321'.
 77  OUTPUT-TEXT                  PIC       X(24)
     VALUE LOW-VALUES.
 77  COMPARE-TEXT                 PIC       X(24)
     VALUE LOW-VALUES.
 77  CIPHER-PROCESSING-RULE       PIC       X(08)
     VALUE 'CUSP    '.
 77  KEY-FORM                     PIC       X(08)
     VALUE 'OP      '.
 77  KEY-LENGTH                   PIC       X(08)
```

```
     VALUE 'SINGLE  '.
 77  KEY-TYPE-1                    PIC       X(08)
     VALUE 'DATA    '.
 77  KEY-TYPE-2                    PIC       X(08)
     VALUE '        '.
 77  ICV                           PIC       X(08)
     VALUE LOW-VALUES.
 77  PAD                           PIC       X(01)
     VALUE LOW-VALUES.
************* DEFINE SAPI INPUT/OUTPUT PARAMETERS ************
 01  SAPI-REC.
     05  RETURN-CODE-S             PIC       9(08) COMP.
     05  REASON-CODE-S             PIC       9(08) COMP.
     05  EXIT-DATA-LENGTH-S        PIC       9(08) COMP.
     05  EXIT-DATA-S               PIC       X(04).
     05  KEK-KEY-ID-1-S            PIC       X(64)
         VALUE LOW-VALUES.
     05  KEK-KEY-ID-2-S            PIC       X(64)
         VALUE LOW-VALUES.
     05  DATA-KEY-ID-S             PIC       X(64)
         VALUE LOW-VALUES.
     05  NULL-KEY-ID-S             PIC       X(64)
         VALUE LOW-VALUES.
     05  KEY-FORM-S                PIC       X(08).
     05  KEY-LENGTH-S              PIC       X(08).
     05  DATA-KEY-TYPE-S           PIC       X(08).
     05  NULL-KEY-TYPE-S           PIC       X(08).
     05  TEXT-LENGTH-S             PIC       9(08) COMP.
     05  TEXT-S                    PIC       X(24).
     05  ICV-S                     PIC       X(08).
     05  PAD-S                     PIC       X(01).
     05  CPHR-TEXT-S               PIC       X(24).
     05  COMP-TEXT-S               PIC       X(24).
     05  RULE-ARRAY-COUNT-S        PIC       9(08) COMP.
     05  RULE-ARRAY-S.
         10  RULE-ARRAY            PIC       X(08).
     05  CHAINING-VECTOR-S         PIC       X(18).
*****************************************************************
 PROCEDURE DIVISION.
*****************************************************************
 MAIN-RTN.
*************  CALL KEY GENERATE  ***************************
     MOVE 0               TO   EXIT-DATA-LENGTH-S.
     MOVE KEY-FORM        TO   KEY-FORM-S.
     MOVE KEY-LENGTH      TO   KEY-LENGTH-S.
     MOVE KEY-TYPE-1      TO   DATA-KEY-TYPE-S.
     MOVE KEY-TYPE-2      TO   NULL-KEY-TYPE-S.
     CALL 'CSFKGN'   USING RETURN-CODE-S
                             REASON-CODE-S
                             EXIT-DATA-LENGTH-S
                             EXIT-DATA-S
                             KEY-FORM-S
                             KEY-LENGTH-S
                             DATA-KEY-TYPE-S
                             NULL-KEY-TYPE-S
                             KEK-KEY-ID-1-S
                             KEK-KEY-ID-2-S
                             DATA-KEY-ID-S
                             NULL-KEY-ID-S.
     IF RETURN-CODE-S NOT = 0 OR
        REASON-CODE-S NOT = 0 THEN
        DISPLAY '*** KEY-GENERATE ***'
        DISPLAY '*** RETURN-CODE = ' RETURN-CODE-S
        DISPLAY '*** REASON-CODE = ' REASON-CODE-S
     ELSE
        MOVE 24                   TO  TEXT-LENGTH-S
        MOVE INPUT-TEXT           TO  TEXT-S
        MOVE 1                   TO  RULE-ARRAY-COUNT-S
        MOVE CIPHER-PROCESSING-RULE TO RULE-ARRAY-S
        MOVE LOW-VALUES          TO  CHAINING-VECTOR-S
        MOVE ICV             TO   ICV-S.
        MOVE PAD             TO   PAD-S.
*************  CALL ENCIPHER ************************************
        CALL 'CSFENC' USING    RETURN-CODE-S
                                 REASON-CODE-S
                                 EXIT-DATA-LENGTH-S
                                 EXIT-DATA-S
                                 DATA-KEY-ID-S
                                 TEXT-LENGTH-S
                                 TEXT-S
                                 ICV-S
                                 RULE-ARRAY-COUNT-S
                                 RULE-ARRAY-S
                                 PAD-S
                                 CHAINING-VECTOR-S
                                 CPHR-TEXT-S
        IF RETURN-CODE-S NOT = 0 OR
           REASON-CODE-S NOT = 0 THEN
           DISPLAY '*** ENCIPHER ***'
           DISPLAY '*** RETURN-CODE = ' RETURN-CODE-S
           DISPLAY '*** REASON-CODE = ' REASON-CODE-S
        ELSE
*************  CALL DECIPHER ************************************
           CALL 'CSFDEC' USING RETURN-CODE-S
                                 REASON-CODE-S
                                 EXIT-DATA-LENGTH-S
                                 EXIT-DATA-S
                                 DATA-KEY-ID-S
                                 TEXT-LENGTH-S
                                 CPHR-TEXT-S
                                 ICV-S
                                 RULE-ARRAY-COUNT-S
                                 RULE-ARRAY-S
                                 CHAINING-VECTOR-S
                                 COMP-TEXT-S
           IF RETURN-CODE-S  NOT = 0 OR
              REASON-CODE-S  NOT = 0 THEN
              DISPLAY '*** DECIPHER ***'
              DISPLAY '*** RETURN-CODE = ' RETURN-CODE-S
              DISPLAY '*** REASON-CODE = ' REASON-CODE-S
          ELSE
              IF COMP-TEXT-S = TEXT-S THEN
                DISPLAY '*** DECIPHERED TEXT = PLAIN TEXT ***'
              ELSE
                DISPLAY '*** DECIPHERED TEXT ê= PLAIN TEXT ***'.
     DISPLAY '*** TEST PROGRAM ENDED ***'
     STOP RUN.
```

Assembler H

```
         TITLE 'SAMPLE ENCIPHER/DECIPHER S/370 PROGRAM.'
*=====================================================================*
*       SYSTEM/370 ASSEMBLER H EXAMPLE                                *
*                                                                     *
*=====================================================================*
         SPACE
SAMPLE   START 0
         DS    0H
         STM   14,12,12(13)     SAVE REGISTERS
         BALR  12,0             USE R12 AS BASE REGISTER
         USING *,12             PROVIDE SAVE AREA FOR SUBROUTINE
         LA    14,SAVE          PERFORM SAVE AREA CHAINING
         ST    13,4(14)           "
         ST    14,8(13)           "
         LR    13,14              "
*
         CALL  CSFKGN,(RETCD,                                          *
               RESCD,                                                  *
               EXDATAL,                                                *
               EXDATA,                                                 *
               KEY_FORM,                                               *
               KEY_LEN,                                                *
               KEYTYP1,                                                *
               KEYTYP2,                                                *
               KEK_ID1,                                                *
               KEK_ID2,                                                *
               DATA_ID,                                                *
               NULL_ID)
         CLC   RETCD,=F'0'      CHECK RETURN CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
         CLC   RESCD,=F'0'      CHECK REASON CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
*
* CALL ENCIPHER WITH THE KEY JUST GENERATED
*  OPERATIONAL FORM
*
         MVC   RULEAC,=F'1'           SET RULE ARRAY COUNT
         MVC   RULEA,=CL8'CUSP    '   BUILD RULE ARRAY
         CALL CSFENC,(RETCD,                                           *
               RESCD,                                                  *
               EXDATAL,                                                *
               EXDATA,                                                 *
               DATA_ID,                                                *
               TEXTL,                                                  *
               TEXT,                                                   *
               ICV,                                                    *
               RULEAC,                                                 *
               RULEA,                                                  *
               PAD_CHAR,                                               *
               OCV,                                                    *
               CIPHER_TEXT)
         CLC   RETCD,=F'0'      CHECK RETURN CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
         CLC   RESCD,=F'0'      CHECK REASON CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
         CALL CSFDEC,(RETCD,                                           *
               RESCD,                                                  *
               EXDATAL,                                                *
               EXDATA,                                                 *
               DATA_ID,                                                *
               TEXTL,                                                  *
               CIPHER_TEXT,                                            *
               ICV,                                                    *
               RULEAC,                                                 *
               RULEA,                                                  *
               OCV,                                                    *
               NEW_TEXT)
         CLC   RETCD,=F'0'      CHECK RETURN CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
         CLC   RESCD,=F'0'      CHECK REASON CODE
         BNE   BACK             OUTPUT RETURN/REASON CODE AND STOP
*
COMPARE  EQU   *                         COMPARE START AND END TEXT
         CLC   TEXT,NEW_TEXT
         BE    GOODENC
         WTO   'DECIPHERED TEXT DOES NOT MATCH STARTING TEXT'
         B      BACK
GOODENC  WTO   'DECIPHERED TEXT MATCHES STARTING TEXT'
*
*
         WTO   'TEST PROGRAM TERMINATING'
         B     RETURN
*
*----------------------------------------------------
* CONVERT RETURN/REASON CODES FROM BINARY TO EBCDIC
*----------------------------------------------------
BACK     DS    0F                OUTPUT RETURN & REASON CODE
         L     5,RETCD           LOAD RETURN CODE
         L     6,RESCD           LOAD REASON CODE
         CVD   5,BCD1            CONVERT TO PACK-DECIMAL
         CVD   6,BCD2
         UNPK  ORETCD,BCD1       CONVERT TO EBCDIC
         UNPK  ORESCD,BCD2
         OI    ORETCD+7,X'F0'    CORRECT LAST DIGIT
         OI    ORESCD+7,X'F0'
*
         MVC   ERROUT+21(4),ORETCD
         MVC   ERROUT+41(4),ORESCD
ERROUT   WTO   'ERROR CODE =     , REASON CODE =     '
RETURN   EQU   *
         L     13,4(13)          SAVE AREA RESTORATION
         MVC   16(4,13),RETCD    SAVE RETURN CODE
         LM    14,12,12(13)
         BR    14                RETURN TO CALLER
*
BCD1     DS    D                 CONVERT TO BCD TEMP AREA
BCD2     DS    D                 CONVERT TO BCD TEMP AREA
ORETCD   DS    CL8'0'            OUTPUT RETURN CODE
ORESCD   DS    CL8'0'            OUTPUT REASON CODE
*
KEY_FORM DC    CL8'OP      '     KEY FORM
KEY_LEN  DC    CL8'SINGLE  '     KEY LENGTH
KEYTYP1  DC    CL8'DATA    '     KEY TYPE 1
KEYTYP2  DC    CL8'        '     KEY TYPE 2
TEXT     DC    C'ABCDEFGHIJKLMNOPQRSTUV0987654321'
TEXTL    DC    F'32'             TEXT LENGTH
CIPHER_TEXT DC CL32' '
NEW_TEXT DC    CL32' '
DATA_ID  DC    XL64'00'          DATA KEY TOKEN
NULL_ID  DC    XL64'00'          NULL KEY TOKEN - UNFILLED
KEK_ID1  DC    XL64'00'          KEK1 KEY TOKEN
KEK_ID2  DC    XL64'00'          KEK2 KEY TOKEN
RETCD    DS    F'0'              RETURN CODE
RESCD    DS    F'0'              REASON CODE
EXDATAL  DC    F'0'              EXIT DATA LENGTH
EXDATA   DS    0C                EXIT DATA
RULEA    DS    1CL8              RULE ARRAY
RULEAC   DS    F'0'              RULE ARRAY COUNT
ICV      DC    XL8'00'           INITIAL CHAINING VECTOR
OCV      DC    XL18'00'          OUTPUT CHAINING VECTOR
PAD_CHAR DC    F'0'              PAD CHARACTER
SAVE     DS    18F               SAVE REGISTER AREA
         END   SAMPLE
```

PL/1

```
/********************************************************************/
/*                                                                  */
/* Sample program to call the one-way hash service to generate      */
/* the SHA-1 hash of the input text and call digital signature      */
/* generate with an RSA key using the ISO 9796 text formatting. The */
/* RSA key token is built from supplied data and imported for the   */
/* signature generate service to use.                               */
/*                                                                  */
/*      INPUT:  TEXT          Message digest to be signed           */
/*                                                                  */
/*      OUTPUT: SIGNATURE_LENGTH  Length of the signature in bytes  */
/*                                Written to a dataset.             */
/*                                                                  */
/*              SIGNATURE     Signature for hash.  Written to a     */
/*                            dataset.                              */
/*                                                                  */
/********************************************************************/
DSIGEXP:PROCEDURE( TEXT ) OPTIONS( MAIN );

/* Declarations - Parameters                                        */

DCL TEXT             CHAR( 64 ) VARYING;

/* Declarations - API parameters                                    */

DCL CHAINING_VECTOR_LENGTH  FIXED BINARY( 31, 0 ) INIT( 128 );
DCL CHAINING_VECTOR         CHAR( 128 );
DCL DUMMY_KEK               CHAR( 64 );
DCL EXIT_DATA               CHAR( 4 );
DCL EXIT_LEN                FIXED BINARY( 31, 0 ) INIT( 0 );

DCL HASH                    CHAR( 20 );
DCL HASH_LENGTH             FIXED BINARY( 31, 0 ) INIT( 20 );

DCL INTERNAL_PKA_TOKEN      CHAR( 1024 );
DCL INTERNAL_PKA_TOKEN_LENGTH FIXED BINARY( 31, 0 );

DCL KEY_VALUE_STRUCTURE     CHAR(139)
                         INIT(( '02000040000300408000000000000000'X ||
                                '01AE28DA4606D885EB7E0340D6BAAC51'X ||
                                '991C0CD0EAE835AFD9CFF3CD7E7EA741'X ||
                                '41DADD24A6331BEDF41A6626522CCF15'X ||
                                '767D167D01A16F970100010252BDAD42'X ||
                                '52BDAD425A8C6045D41AFAF746BEBD5F'X ||
                                '085D574FCD9C07F0B38C2C45017C2A1A'X ||
                                'B919ED2551350A76606BFA6AF2F1609A'X ||
                                '00A0A48DD719A55E9CA801'X ));
DCL KEY_VALUE_LENGTH        FIXED BINARY( 31, 0 ) INIT( 139 );

DCL OWH_TEXT                CHAR( 64 );

DCL PKA_KEY_TOKEN           CHAR( 1024 );
DCL PKA_TOKEN_LENGTH        FIXED BINARY( 31, 0 );

DCL PRIVATE_NAME            CHAR( 64 ) INIT( 'PL1.EXAMPLE.FOR.APG' );
DCL PRIVATE_NAME_LENGTH     FIXED BINARY( 31, 0 ) INIT( 0 );

DCL RETURN_CODE             FIXED BINARY( 31, 0 ) INIT( 0 );
DCL REASON_CODE             FIXED BINARY( 31, 0 ) INIT( 0 );

DCL RESERVED_FIELD_LENGTH   FIXED BINARY( 31, 0 ) INIT( 0 );
DCL RESERVED_FIELD          CHAR( 1 );

DCL RULE_ARY_CNT_DSG        FIXED BINARY( 31, 0 ) INIT( 1 );
DCL RULE_ARY_CNT_PKB        FIXED BINARY( 31, 0 ) INIT( 1 );
DCL RULE_ARY_CNT_PKI        FIXED BINARY( 31, 0 ) INIT( 0 );
DCL RULE_ARY_CNT_OWH        FIXED BINARY( 31, 0 ) INIT( 2 );
DCL RULE_ARY_DSG            CHAR( 8 ) INIT( 'ISO-9796' );
DCL RULE_ARY_PKB            CHAR( 8 ) INIT( 'RSA-PRIV' );
DCL RULE_ARY_PKI            CHAR( 8 );
DCL RULE_ARY_OWH            CHAR( 16 ) INIT( 'SHA-1   ONLY    ' );

DCL SIGNATURE_LENGTH        FIXED BINARY( 31, 0 );
DCL SIGNATURE               CHAR( 128 );
DCL SIG_BIT_LENGTH          FIXED BINARY( 31, 0 );

DCL TEXT_LENGTH             FIXED BINARY( 31, 0 );

/* Declarations - Files and entry points                            */
```

```
DCL SYSPRINT  FILE OUTPUT;
DCL SIGOUT    FILE RECORD OUTPUT;

DCL CSNDPKB   ENTRY EXTERNAL OPTIONS( ASM, INTER );
DCL CSNDPKI   ENTRY EXTERNAL OPTIONS( ASM, INTER );
DCL CSNBOWH   ENTRY EXTERNAL OPTIONS( ASM, INTER );
DCL CSNDDSG   ENTRY EXTERNAL OPTIONS( ASM, INTER );

/* Declarations - Internal variables                                */

DCL DSG_HEADER       CHAR( 32 )
                           INIT( '* DIGITAL SIGNATURE GENERATION *' );
DCL FILE_OUT_LINE    CHAR( 128 );
DCL OWH_HEADER       CHAR( 16 )
                           INIT( '* ONE WAY HASH *' );
DCL PKB_HEADER       CHAR( 16 )
                           INIT( '* PKA TOKEN BUILD *' );
DCL PKI_HEADER       CHAR( 16 )
                           INIT( '* PKA TOKEN IMPORT *' );
DCL RC_STRING        CHAR( 14 ) INIT( 'RETURN CODE = ' );
DCL RS_STRING        CHAR( 14 ) INIT( 'REASON CODE = ' );
DCL SIG_STRING       CHAR( 12 ) INIT( 'SIGNATURE = ' );
DCL SIG_LEN_STRING   CHAR( 26 ) INIT( 'SIGNATURE LENGTH(BYTES) = ' );

/* Declarations - Built-in functions                                */

DCL (SUBSTR, LENGTH) BUILTIN;

/********************************************************************/
/* Call one-way hash to get the SHA-1 hash of the text.             */
/********************************************************************/
TEXT_LENGTH = LENGTH( TEXT );
OWH_TEXT = SUBSTR( TEXT, 1, TEXT_LENGTH );

CALL CSNBOWH( RETURN_CODE,
              REASON_CODE,
              EXIT_LEN,
              EXIT_DATA,
              RULE_ARY_CNT_OWH,
              RULE_ARY_OWH,
              TEXT_LENGTH,
              OWH_TEXT,
              CHAINING_VECTOR_LENGTH,
              CHAINING_VECTOR,
              HASH_LENGTH,
              HASH );

PUT SKIP LIST( OWH_HEADER );
PUT SKIP LIST( RC_STRING || RETURN_CODE );
PUT SKIP LIST( RS_STRING || REASON_CODE );

/********************************************************************/
/* Create the PKA RSA private external token.                       */
/********************************************************************/
IF RETURN_CODE = 0 THEN
  DO;

  PKA_TOKEN_LENGTH = 1024;

  CALL CSNDPKB( RETURN_CODE,
                REASON_CODE,
                EXIT_LEN,
                EXIT_DATA,
                RULE_ARY_CNT_PKB,
                RULE_ARY_PKB,
                KEY_VALUE_LENGTH,
                KEY_VALUE_STRUCTURE,
                PRIVATE_NAME_LENGTH,
                PRIVATE_NAME,
                RESERVED_FIELD_LENGTH,
                RESERVED_FIELD,
                RESERVED_FIELD_LENGTH,
                RESERVED_FIELD,
                RESERVED_FIELD_LENGTH,
                RESERVED_FIELD,
                RESERVED_FIELD_LENGTH,
                RESERVED_FIELD,
                RESERVED_FIELD_LENGTH,
                RESERVED_FIELD,
                PKA_TOKEN_LENGTH,
                PKA_KEY_TOKEN );

  PUT SKIP LIST( PKB_HEADER );
  PUT SKIP LIST( RC_STRING || RETURN_CODE );
  PUT SKIP LIST( RS_STRING || REASON_CODE );

  END;

/********************************************************************/
/* Import the clear RSA private external token.                     */
/********************************************************************/
IF RETURN_CODE = 0 THEN
  DO;

  INTERNAL_PKA_TOKEN_LENGTH = 1024;

  CALL CSNDPKI( RETURN_CODE,
                REASON_CODE,
                EXIT_LEN,
                EXIT_DATA,
                RULE_ARY_CNT_PKI,
                RULE_ARY_PKI,
                PKA_TOKEN_LENGTH,
                PKA_KEY_TOKEN,
                DUMMY_KEK,
                INTERNAL_PKA_TOKEN_LENGTH,
                INTERNAL_PKA_TOKEN );

  PUT SKIP LIST( PKI_HEADER );
  PUT SKIP LIST( RC_STRING || RETURN_CODE );
  PUT SKIP LIST( RS_STRING || REASON_CODE );

  END;
/********************************************************************/
/* Call digital signature generate.                                 */
/********************************************************************/
IF RETURN_CODE = 0 THEN
  DO;

  SIGNATURE_LENGTH = 128;

  CALL CSNDDSG( RETURN_CODE,
                REASON_CODE,
                EXIT_LEN,
                EXIT_DATA,
                RULE_ARY_CNT_DSG,
                RULE_ARY_DSG,
                INTERNAL_PKA_TOKEN_LENGTH,
                INTERNAL_PKA_TOKEN,
                HASH_LENGTH,
                HASH,
                SIGNATURE_LENGTH,
                SIG_BIT_LENGTH,
                SIGNATURE );

  PUT SKIP LIST( DSG_HEADER );
  PUT SKIP LIST( RC_STRING || RETURN_CODE );
  PUT SKIP LIST( RS_STRING || REASON_CODE );

  IF RETURN_CODE = 0 THEN
    DO;

    /*****************************************************************/
    /* Write the signature and its length to the output file.       */
    /*****************************************************************/
    FILE_OUT_LINE = SIG_LEN_STRING || SIGNATURE_LENGTH;
    WRITE FILE(SIGOUT) FROM( FILE_OUT_LINE );
    FILE_OUT_LINE = SIG_STRING || SIGNATURE;
    WRITE FILE(SIGOUT) FROM( FILE_OUT_LINE );
    END;

  END;

END DSIGEXP;
```

Appendix E. Using ICSF with BSAFE

ICSF works in conjunction with RSA Security, Inc.'s BSAFE toolkit (BSAFE 3.1 or later). If you are currently using applications developed with BSAFE, you may want to take advantage of the increased security and performance available with the Cryptographic Coprocessor Feature and ICSF.

Through BSAFE 3.1 you can access the ICSF services to:

- Compute message digests or hashes
- Generate random numbers
- Encipher and decipher data using the DES algorithm
- Generate and verify RSA digital signatures

Some BSAFE Basics

BSAFE has many algorithm information types (called AIs). Many of the AIs can perform several cryptographic functions. For this reason, you must specify the algorithmic method (AM) to be used by supplying a chooser. If the cryptographic function requires a key, you supply key information to the BSAFE application with a key information (KI) type. For the most current information on the BSAFE user interface and a complete description of algorithm information types, algorithm methods, choosers, and key information types, refer to *BSAFE User's Manual* and *BSAFE Library Reference Manual*.

Computing Message Digests and Hashes

MD5 and SHA1 hashing are both available from ICSF via BSAFE. If your BSAFE application uses the AM_MD5 or the AM_SHA algorithm methods, you can add a couple of BSAFE function calls and the application will use ICSF and the Cryptographic Coprocessor Feature instead of the BSAFE algorithm method.

The following list shows BSAFE AI types with choosers that may include AM_MD5:

- AI_MD5
- AI_MD5_BER
- AI_MD5WithDES_CBCPad
- AI_MD5WithDES_CBCPadBER
- AI_MD5WithRC2_CBCPad
- AI_MD5WithRC2_CBCPadBER
- AI_MD5WithRSAEncryption
- AI_MD5WithRSAEncryptionBER
- AI_MD5WithXOR
- AI_MD5WithXOR_BER

The following list shows BSAFE AI types with choosers that may include AM_SHA:

- AI_SHA1
- AI_SHA1_BER
- AI_SHA1WithDES_CBCPad
- AI_SHA1WithDES_CBCPadBER

Generating Random Numbers

If your BSAFE application uses the algorithm method AM_MD5_RANDOM, you can add a chooser definition containing the algorithm method AM_HW_RANDOM (new with BSAFE 3.1) and a couple of BSAFE function calls and your program can use ICSF and the Cryptographic Coprocessor Feature to generate random numbers instead of the BSAFE algorithm method.

BSAFE 3.1 provides a new algorithm information type, AI_HWRandom. You need to set your random number generation object with AI_HWRandom, and initialize the object with a chooser containing AM_HW_RANDOM, in order to use ICSF with the Cryptographic Coprocessor Feature for generating random numbers. You do not, however, have to make a B_RandomUpdate call, since the S/390 and IBM @server zSeries cryptographic solution does not require a seed.

The only AI type with choosers that may include AM_HW_RANDOM is AI_HWRandom.

Encrypting and Decrypting with DES

If your BSAFE application uses either the AM_DES_CBC_ENCRYPT or the AM_DES_CBC_DECRYPT algorithm methods, you can add a chooser containing the algorithm methods AM_TOKEN_DES_CBC_ENCRYPT and/or AM_TOKEN_DES_CBC_DECRYPT (both new with BSAFE 3.1) and a couple of BSAFE function calls and your program can use ICSF and the Cryptographic Coprocessor Feature to encrypt and/or decrypt data using the DES algorithm.

For your encryption or decryption key, you can use either a clear key in the form of a KI_8Byte or KI_DES8 or KI_Item (8 bytes long), or a CCA DES Key Token in the form of a KI_TOKEN (64 bytes long). KI_TOKEN is a new key information type in BSAFE 3.1.

The following list shows BSAFE AI types with choosers that may include either AM_TOKEN_DES_CBC_ENCRYPT, AM_TOKEN_DES_CBC_DECRYPT, or both:

- AI_DES_CBC_BSAFE1
- AI_DES_CBC_IV8
- AI_DES_CBCPadBER
- AI_DES_CBCPadIV8
- AI_DES_CBCPadPEM
- AI_MD5WithDES_CBCPad
- AI_MD5WithDES_CBCPadBER
- AI_SHA1WithDES_CBCPad
- AI_SHA1WithDES_CBCPadBER

Generating and Verifying RSA Digital Signatures

You can use algorithm method AM_TOKEN_RSA_PRV_ENCRYPT with AM_MD5 or AM_SHA to have ICSF and the Cryptographic Coprocessor Feature generate RSA digital signatures. To verify the RSA digital signature using the S/390 or IBM @server zSeries cryptographic solution, you can use AM_TOKEN_RSA_PUB_DECRYPT (with AM_MD5 or AM_SHA). Your BSAFE application must contain a couple of new BSAFE function calls to access the S/390 and IBM @server zSeries services. AM_TOKEN_RSA_PRV_ENCRYPT and AM_TOKEN_RSA_PUB_DECRYPT are new in BSAFE 3.1. For more information, see “Using the New Function Calls in Your BSAFE Application” on page 413.

For signature generation, you can use either a clear private key in the form of a KI_PKCS_RSAPrivate or a CCA RSA private key token in the form of a KI_TOKEN. For signature verification, you can use either a public RSA key in the form of a KI_RSAPublic or a CCA RSA public key token in the form of a KI_TOKEN.

KI_TOKEN is a new key information type in BSAFE. For more information about KI_TOKEN, see “Using the BSAFE KI_TOKEN” on page 415.

The following list shows BSAFE AI types with choosers that may include AM_TOKEN_RSA_PRV_ENCRYPT:
- AI_MD5WithRSAEncryption
- AI_MD5WithRSAEncryptionBER
- AI_SHA1WithRSAEncryption
- AI_SHA1WithRSAEncryptionBER

The following list shows BSAFE AI types with choosers that may include AM_TOKEN_RSA_PUB_DECRYPT:
- AI_MD5WithRSAEncryption
- AI_SHA1WithRSAEncryption

Encrypting and Decrypting with RSA

You can use algorithm method AM_TOKEN_RSA_ENCRYPT to have ICSF encrypt a symmetric key (or other string of 48 bytes or fewer). To decrypt the string using ICSF, you can use AM_TOKEN_RSA_CRT_DECRYPT. You’ll need a couple of new BSAFE function calls to access the S/390 and IBM @server zSeries services (see “Using the New Function Calls in Your BSAFE Application”.

To encrypt a string, you can use either a public key in the form KI_RSAPublic or a CCA RSA public key token in the form of a KI_TOKEN.

To decrypt a string, you can use either a private key in the form KI_PKCS_RSAPrivate or a CCA RSA private key token in the form of a KI_TOKEN.

Using the New Function Calls in Your BSAFE Application

To have your BSAFE application access the ICSF, S/390, and IBM @server zSeries Cryptographic Coprocessor Feature services, you need to add several new elements to your program. These elements are explained with examples in the steps that follow.

1. At the beginning of your program, declare one or more session choosers and also the hardware table list. For information about choosers and the hardware table list, see *BSAFE User’s Manual*.

   ```
   /*-------------------------------------------------------------*
    * SESSION_CHOOSER will replace OLD_CHOOSER.                   *
    *-------------------------------------------------------------*/
   B_ALGORITHM_METHOD **SESSION_CHOOSER = NULL_PTR;


   /*-------------------------------------------------------------*
    * CCA_VTABLE is a vector table of functions that will be      *
    * substituted for BSAFE equivalents.  It is supplied by IBM   *
    * and will be loaded into your application when you invoke    *
    * QueryCrypto.                                                *
    *-------------------------------------------------------------*/
   HW_TABLE_LIST CCA_VTABLE = (HW_TABLE_LIST)NULL_PTR;
   ```

2. Declare a tag list. The content of the tag list is supplied by BSAFE at the B_CreateSessionChooser call, which is discussed in a later step.

   ```
   unsigned char **taglist = (unsigned char **)NULL_PTR;
   ```

3. For random number generation, DES encryption or decryption or RSA encryption or decryption, you need to define and declare an additional chooser

wherever your current chooser is defined and declared. For instance, suppose your application is doing an RSA encryption, and OLD_CHOOSER is defined as follows:

```
/*--------------------------------------------------------------*
 * OLD_CHOOSER is used for this application when ICSF and       *
 * the crypto hardware is not available.                        *
 *--------------------------------------------------------------*/
B_ALGORITHM_METHOD *OLD_CHOOSER[] = {
  &AM_SHA,
  &AM_RSA_ENCRYT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

/*--------------------------------------------------------------*
 * ICSF_CHOOSER is a 'skeleton' for SESSION_CHOOSER.            *
 * SESSION_CHOOSER will be used for this application if         *
 * ICSF and the crypto hardware are not available.              *
 *--------------------------------------------------------------*/
B_ALGORITHM_METHOD *ICSF_CHOOSER[]  = {
  &AM_SHA,
  &AM_TOKEN_RSA_PUB_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

4. At the beginning of the main function in your application, add a call to the ICSF QueryCrypto function followed by a conditional call to the BSAFE B_CreateSessionChooser function.

```
/*-------------------------------------------------------------*
 * Check for the existence of crypto hardware.  If it's there, *
 * QueryCrypto will supply CCA_VTABLE                          *
 *-------------------------------------------------------------*/
if ((status = QueryCrypto(CRYPTO_Q_DES_AND_RSA,&CCA_VTABLE)) == 0)
/*----------------------------------------------*
                 * B_CreateSessionChooser will replace the     *
                 * BSAFE software functions with their CCA     *
                 * hardware equivalents.                       *
                 *                                             *
                 * Note that the last three parameters are not *
                 * used with CCA                               *
                 *---------------------------------------------*/
  if ((status = B_CreateSessionChooser(ICSF_CHOOSER,
                                      &SESSION_CHOOSER,
                                       CCA_VTABLE,
                                       (ITEM *)NULL_PTR,
                                       (POINTER *)NULL_PTR,
                                         &taglist)) != 0)
    break;
```

5. Set up the conditions under which any alternate choosers are used to initialize the appropriate algorithm object. For information about initializing algorithm objects, see *BSAFE User's Manual*.

```
/*-------------------------------------------------------*
 * Initialize the algorithm object with the appropriate  *
 * chooser.                                              *
 *-------------------------------------------------------*/
if (SESSION_CHOOSER != NULL_PTR)
  if ((status = B_xxxxxxInit
       (xxxxxxObject,SESSION_CHOOSER,
        (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
  else ;
else
  if ((status = B_xxxxxxInit
       (xxxxxxObject,OLD_CHOOSER,
        (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
  else ;
```

6. When your application no longer needs the session chooser, program a call to the BSAFE B_FreeSessionChooser function.

   ```
   if (SESSION_CHOOSER != NULL_PTR)
     B_FreeSessionChooser(&SESSION_CHOOSER,&taglist);
   ```

Using the BSAFE KI_TOKEN

Those ICSF functions that require a key, like encipher and decipher, expect the key in the form of a CCA token. If you already have a CCA token, perform the following steps before you try to set your algorithm object. For information about how to perform the following tasks, see *BSAFE User's Manual* and *BSAFE Library Reference Manual*.

1. Create a key object.
2. Declare a KEY_TOKEN_INFO and fill it in.

   KEY_TOKEN_INFO is defined as follows in the *BSAFE User's Manual*:

   ```
   typedef struct {
     ITEM manufacturerID;
     ITEM internalKeyInfo;
   } KEY_TOKEN_INFO;
   ```

   The first ITEM is the address and length of one of the following three strings, depending on the CCA key token type you are using:

   - com.ibm.CCADES
   - com.ibm.CCARSAPublic
   - com.ibm.CCARSAPrivate

   The second ITEM is the address and length of your CCA key token.
3. Set the key information (B_SetKeyInfo) into the key object using the item and a key information type of KI_TOKEN as input.

If you don't already have a CCA token, you can supply a clear key to the function using one of the key information types mentioned in the section discussing the function you are using. BSAFE will convert the key to a CCA token. If you supply a clear BSAFE KI type to one of the ICSF functions, and the function is performed successfully, you can retrieve the key as a CCA token by invoking B_GetKeyInfo with KI_TOKEN as the key information type. A KEY_TOKEN_INFO struct is returned.

ICSF Triple DES via BSAFE

ICSF performs single, double, or triple DES depending on the length of the DES key; if you're using BSAFE to access ICSF triple DES, you should use the algorithm methods AM_TOKEN_DES_CBC_ENCRYPT and AM_TOKEN_DES_CBC_DECRYPT.

If you've already have an ICSF token, follow the instructions in the section titled "Using the BSAFE KI_TOKEN".

If you're using a clear key, follow the same procedure, except use your clear key padded on the right with binary zeroes to a length of 64 as the internalKeyInfo part of your KI_TOKEN_INFO. ICSF will convert your clear key to an internal ICSF key token.

Here's an example:

```
B_KEY_OBJ desKey = (B_KEY_OBJ)NULL_PTR;
KEY_TOKEN_INFO myTokenInfo;
unsigned char myToken[64] = {0};
unsigned char * myTokenP;
unsigned char myDoubleKey[16];   /* Input to this function  *
unsigned char mfgID[] = "com.ibm.CCADES";
unsigned char * mfgIDP;
    .
    .
    .
myTokenP = myToken;
mfgIDP = mfgID;
T_memcpy(myToken,myDoubleKey,sizeof(myDoubleKey));
myTokenInfo.manufacturerID.len = strlen(mfgID);
myTokenInfo.manufacturerID.data = mfgIDP;
myTokenInfo.internalKeyInfo.len = sizeof(myToken);
myTokenInfo.internalKeyInfo.data = myTokenP;

/*  Create a key object. */
if ((status = B_CreateKeyObject (&desKey)) != 0)
   break;

/*  Set the key object.  */
if ((status = B_SetKeyInfo
   (desKey, KI_TOKEN, myTokenInfo )) != 0)
  break;
    .
    .
    .
```

Retrieving ICSF Error Information

When using the ICSF and Cryptographic Coprocessor Feature, Init, Update, and Final calls can result in BSAFE returning a status of BE_HARDWARE (0x020B). When this occurs, you can derive the ICSF return and reason codes by using a new BSAFE operation, B_GetExtendedErrorInfo. For an explanation of the return codes and reason codes, see Appendix A, “ICSF and TSS Return and Reason Codes” on page 335.

A coding example follows.

```
    .
    .
#include "balg.h"
#include "algobj.h"
#include "cca.h"
    .
    .
{
    .
    .
    .
  B_ALGORITHM_OBJECT * aop;
  ITEM * errp;
  unsigned char * algorithmMethod;
  CCA_ERROR_DATA * edp;
  unsigned int CCAreturnCode=0;
```

```
   unsigned int CCAreasonCode=0;
   unsigned char algorithmName[40]={0x00};
     .
     .
     .
   if (status==BE_HARDWARE) {
     B_GetExtendedErrorInfo(aop,errp,algorithmMethod);
     edp = errp->data;
     CCAreturnCode = (unsigned int) edp->returnCode;
     CCAreasonCode = (unsigned int) edp->reasonCode;
   }
     .
     .
  }
```

The prototype for B_GetExtendedErrorInfo is in balg.h, as shown in the example that follows.

```
B_GetExtendedErrorInfo (
B_ALGORITHM_OBJ algorithmObject,  /* in--algorithm object  */
ITEM * errorData,                 /* out--address and length of error data */
POINTER algorithmMethod           /* out--address of faulting AM  */
);
```

Appendix F. Cryptographic Algorithms and Processes

This appendix describes the personal identification number (PIN) formats and algorithms.

PIN Formats and Algorithms

For PIN calculation procedures, see *IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*.

PIN Notation

This section describes various PIN block formats. The following notations describe the contents of PIN blocks:

P = A 4-bit decimal digit that is one digit of the PIN value.

C = A 4-bit hexadecimal control value. The valid values are X'0', X'1', and X'2'.

L = A 4-bit hexadecimal value that specifies the number of PIN digits. The value ranges from 4 to 12, inclusive.

F = A 4-bit field delimiter of value X'F'.

f = A 4-bit delimiter filler that is either P or F, depending on the length of the PIN.

D = A 4-bit decimal padding value. All pad digits in the PIN block have the same value.

X = A 4-bit hexadecimal padding value. All pad digits in the PIN block have the same value.

x = A 4-bit hexadecimal filler that is either P or X, depending on the length of the PIN.

R = A 4-bit hexadecimal random digit. The sequence of R digits can each take a different value.

r = A 4-bit random filler that is either P or R, depending on the length of the PIN.

Z = A 4-bit hexadecimal zero (X'0').

z = A 4-bit zero filler that is either P or Z, depending on the length of the PIN.

S = A 4-bit hexadecimal digit that constitutes one digit of a sequence number.

A = A 4-bit decimal digit that constitutes one digit of a user-specified constant.

PIN Block Formats

This section describes the PIN block formats and assigns a code to each format.

ANSI X9.8

This format is also named ISO format 0, VISA format 1, VISA format 4, and ECI format 1.

```
P1 = CLPPPPffffffffFF

     P2 = ZZZZAAAAAAAAAAAA
```

```
   PIN Block = P1 XOR P2

where C = X'0'
      L = X'4' to X'C'
```

Programming Note: The rightmost 12 digits in P2 are the rightmost 12 digits of the account number for all formats except VISA format 4. For VISA format 4, the rightmost 12 digits in P2 are the leftmost 12 digits of the account number.

ISO Format 1

This format is also named ECI format 4.

```
   PIN Block = CLPPPPrrrrrrrrRR

where C = X'1'
      L = X'4' to X'C'
```

ISO Format 2

```
   PIN Block = CLPPPPffffffffFF

where C = X'2'
      L = X'4' to X'C'
```

VISA Format 2

```
   PIN Block = LPPPPzzDDDDDDDDD

where L = X'4' to X'6'
```

VISA Format 3

This format specifies that the PIN length can be 4-12 digits, inclusive. The PIN starts from the leftmost digit and ends by the delimiter (‘F’), and the remaining digits are padding digits.

An example of a 6-digit PIN:

```
   PIN Block = PPPPPPFXXXXXXXXX
```

IBM 4700 Encrypting PINPAD Format

This format uses the value X'F' as the delimiter for the PIN.

```
   PIN Block = LPPPPffffffffFSS

where L = X'4' to X'C'
```

IBM 3624 Format

This format requires the program to specify the delimiter, X, for determining the PIN length.

```
   PIN Block = PPPPxxxxxxxxXXXX
```

IBM 3621 Format

This format requires the program to specify the delimiter, X, for determining the PIN length.

```
   PIN Block = SSSSPPPPxxxxxxxx
```

ECI Format 2

This format defines the PIN to be 4 digits.

```
   PIN Block = PPPPRRRRRRRRRRRR
```

ECI Format 3

```
   PIN Block = LPPPPzzRRRRRRRRR

where L = X'4' to X'6'
```

PIN Extraction Rules

This section describes the PIN extraction rules for the Encrypted PIN verify and Encrypted PIN translate callable services.

Encrypted PIN Verify Callable Service

The service extracts the customer-entered PIN from the input PIN block according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 1, ISO format 2, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3, the service extracts the PIN according to the length specified in the PIN block.
- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through rule array keywords. If no PIN extraction method is specified in the rule array, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through rule array keywords. If no PIN extraction method is specified in the rule array, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is ECI format 2, the PIN is the leftmost 4 digits.

For the VISA algorithm, if the extracted PIN length is less than 4, the services sets a reason code that indicates that verification failed. If the length is greater than or equal to 4, the service uses the leftmost 4 digits as the referenced PIN.

For the IBM German Banking Pool algorithm, if the extracted PIN length is not 4, the service sets a reason code that indicates that verification failed.

For the IBM 3624 algorithm, if the extracted PIN length is less than the PIN check length, the service sets a reason code that indicates that verification failed.

Clear PIN Generate Alternate Callable Service

The service extracts the customer-entered PIN from the input PIN block according to the following rules:

- This service supports the specification of a PIN extraction method for the 3624 and 3621 PIN block formats through the use of the rule_array keyword. *Rule_array* points to an array of one or two 8-byte elements. The first element in the rule array specifies the PIN calculation method. The second element in the rule array (if specified) indicates the PIN extraction method. Refer to the “Clear PIN Generate Alternate (CSNBCPA)” on page 220 for an explanation of PIN extraction method keywords.

Encrypted PIN Translate Callable Service

The service extracts the customer-entered PIN from the input PIN block according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 1, ISO format 2, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3, and if the specified PIN length is less than 4, the service sets a reason code to reject the operation. If the specified PIN length is greater than 12, the operation proceeds to normal completion with unpredictable contents in the output PIN block. Otherwise, the service extracts the PIN according to the specified length.

- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through rule array keywords. If no PIN extraction method is specified in the rule array, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through rule array keywords. If no PIN extraction method is specified in the rule array, the specified delimiter (padding) determines the PIN length.
- If the input block format is ECI format 2, the PIN is always the leftmost 4 digits.

If the maximum PIN length allowed by the output PIN block is shorter than the extracted PIN, only the leftmost digits of the extracted PIN that form the allowable maximum length are placed in the output PIN block. The PIN length field in the output PIN block, it if exists, specifies the allowable maximum length.

IBM PIN Algorithms

This section describes the IBM PIN generation algorithms, IBM PIN offset generation algorithm, and IBM PIN verification algorithms.

3624 PIN Generation Algorithm

This algorithm generates a n-digit PIN based on an account-related data or person-related data, namely the validation data. The assigned PIN length parameter specifies the length of the generated PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 4-bit assigned PIN length
- A 128-bit PIN-generation key

The service uses the PIN generation key to encipher the validation data. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of the enciphered validation data. The result is an intermediate PIN. The leftmost n digits of the intermediate PIN are the generated PIN, where n is specified by the assigned PIN length.

Figure 9 illustrates the 3624 PIN generation algorithm.

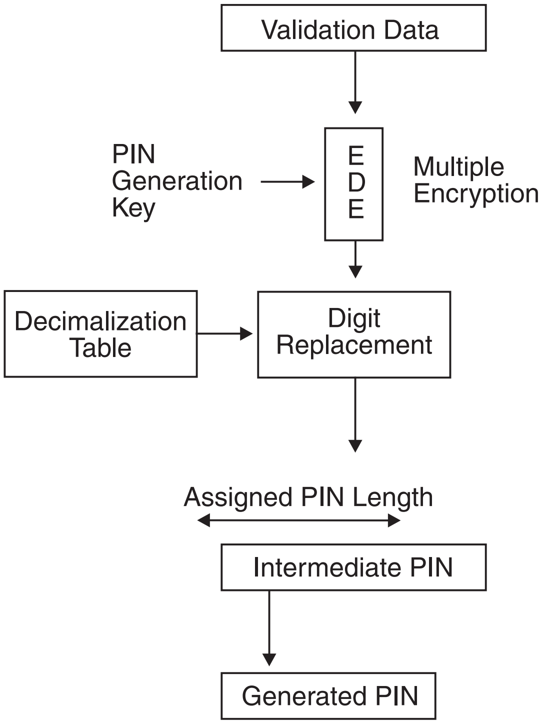


Figure 9. 3624 PIN Generation Algorithm

German Banking Pool PIN Generation Algorithm

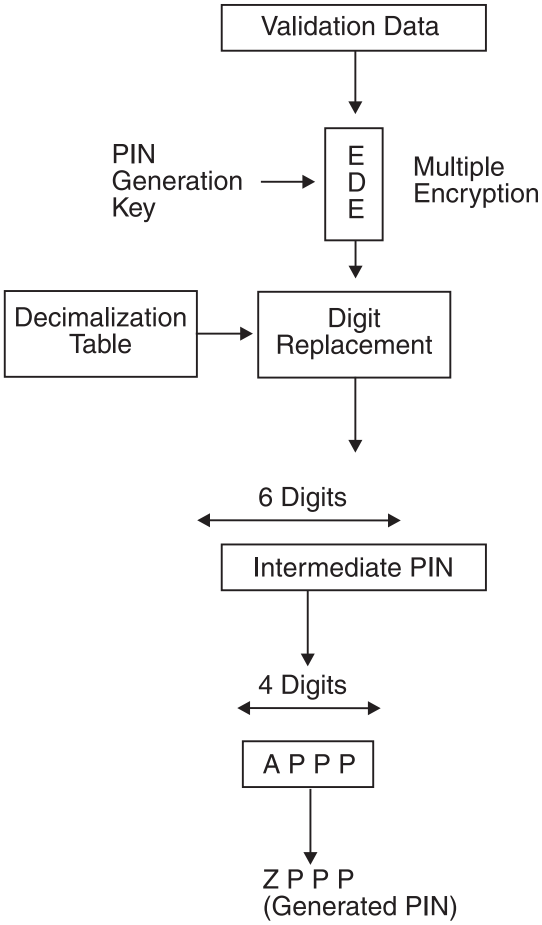
This algorithm generates a 4-digit PIN based on an account-related data or person-related data, namely the validation data.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-generation key

The validation data is enciphered using the PIN generation key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data. The result is an intermediate PIN. The rightmost 4 digits of the leftmost 6 digits of the intermediate PIN are extracted. The leftmost digit of the extracted 4 digits is checked for zero. If the digit is zero, the digit is changed to one; otherwise, the digit remains unchanged. The resulting four digits is the generated PIN.

Figure 10 illustrates the German Banking Pool (GBP) PIN generation algorithm.



If A = 0, then Z = 1; otherwise, Z = A.

Figure 10. GBP PIN Generation Algorithm

PIN Offset Generation Algorithm

To allow the customer to select his own PIN, a PIN offset is used by the IBM 3624 and GBP PIN generation algorithms to relate the customer-selected PIN to the generated PIN.

The PIN offset generation algorithm requires two parameters in addition to those used in the 3624 PIN generation algorithm. They are a customer-selected PIN and a 4-bit PIN check length. The length of the customer-selected PIN is equal to the assigned-PIN length, n.

The 3624 PIN generation algorithm described in the previous section is performed. The offset data value is the result of subtracting (modulo 10) the leftmost n digits of the intermediate PIN from the customer-selected PIN. The modulo 10 subtraction ignores borrows. The rightmost m digits of the offset data form the PIN offset, where m is specified by the PIN check length. Note that n cannot be less than m. To generate a PIN offset for a GBP PIN, m is set to 4 and n is set to 6.

Figure 11 illustrates the PIN offset generation algorithm.

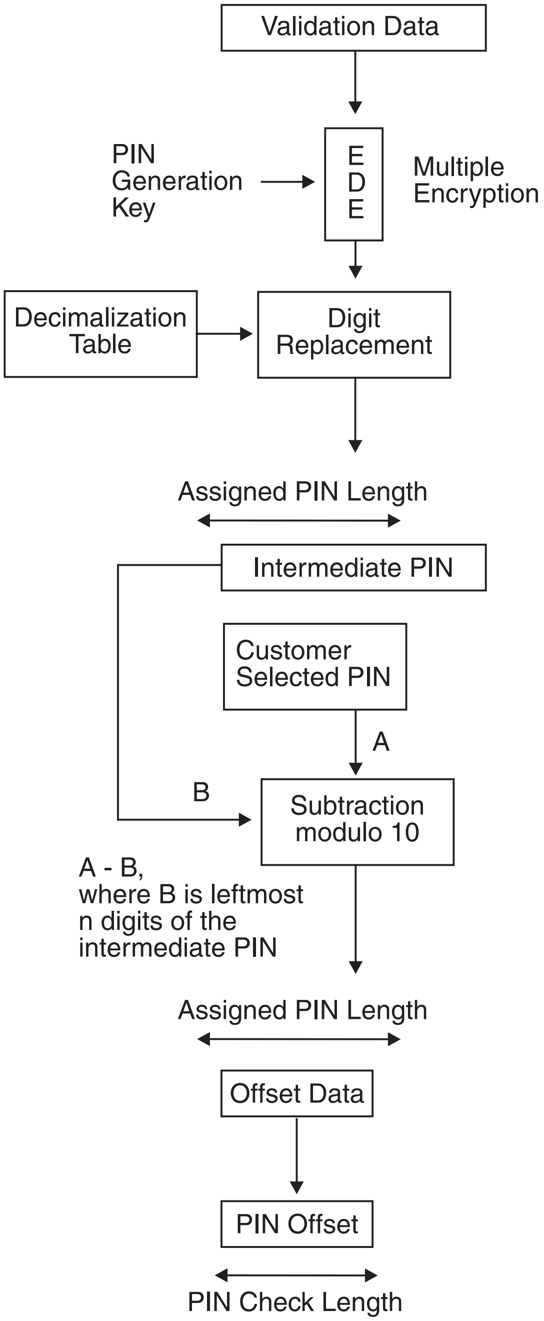


Figure 11. PIN-Offset Generation Algorithm

3624 PIN Verification Algorithm

This algorithm generates an intermediate PIN based on the specified validation data. A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is compared with the corresponding part of the customer-entered PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-verification key
- A 4-bit PIN check length

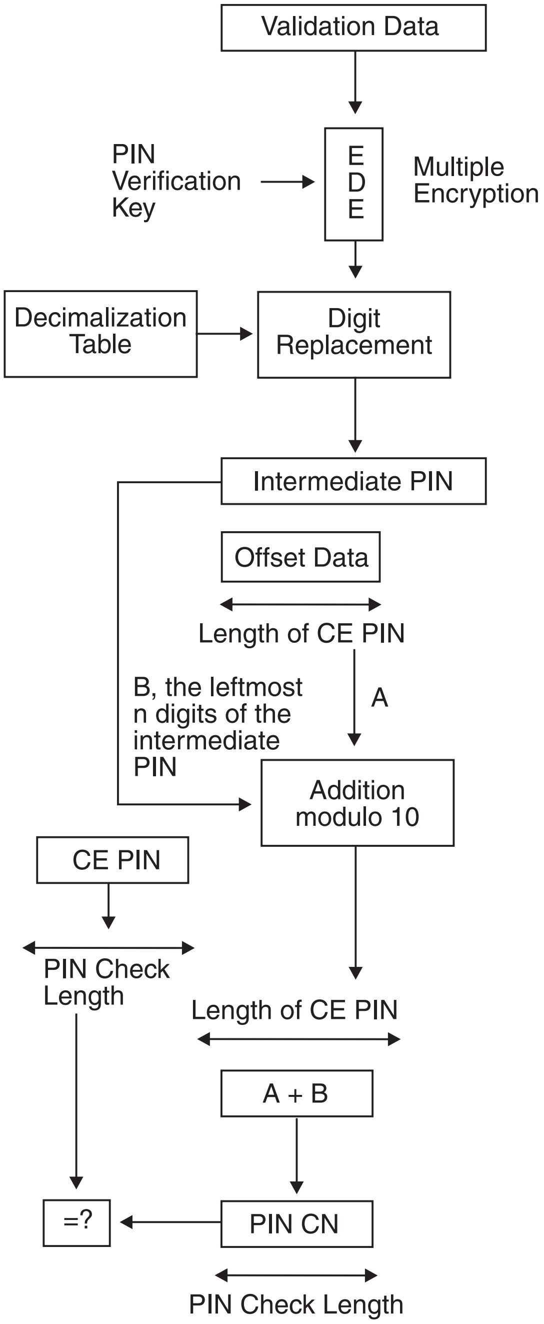
- An offset data
- A customer-entered PIN

The rightmost m digits of the offset data form the PIN offset, where m is the PIN check length.

1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data.
2. The leftmost n digits of the result is added (modulo 10) to the offset data value, where n is the length of the customer-entered PIN. The modulo 10 addition ignores carries.
3. The rightmost m digits of the result of the addition operation form the PIN check number. The PIN check number is compared with the rightmost m digits of the customer-entered PIN. If they match, PIN verification is successful; otherwise, verification is unsuccessful.

When a nonzero PIN offset is used, the length of the customer-entered PIN is equal to the assigned PIN length.

Figure 12 illustrates the PIN verification algorithm.



PIN CN: PIN Check Number
CE PIN: Customer-entered PIN

Figure 12. PIN Verification Algorithm

German Banking Pool PIN Verification Algorithm

This algorithm generates an intermediate PIN based on the specified validation data. A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is extracted. The extracted value may or may not be modified before it compares with the customer-entered PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN verification key
- An offset data
- A customer-entered PIN

The rightmost 4 digits of the offset data form the PIN offset.

1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data.
2. The leftmost 6 digits of the result is added (modulo 10) to the offset data. The modulo 10 addition ignores carries.
3. The rightmost 4 digits of the result of the addition (modulo 10) are extracted.
4. The leftmost digit of the extracted value is checked for zero. If the digit is zero, the digit is set to one; otherwise, the digit remains unchanged. The resulting four digits are compared with the customer-entered PIN. If they match, PIN verification is successful; otherwise, verification is unsuccessful.

Figure 13 illustrates the GBP PIN verification algorithm.

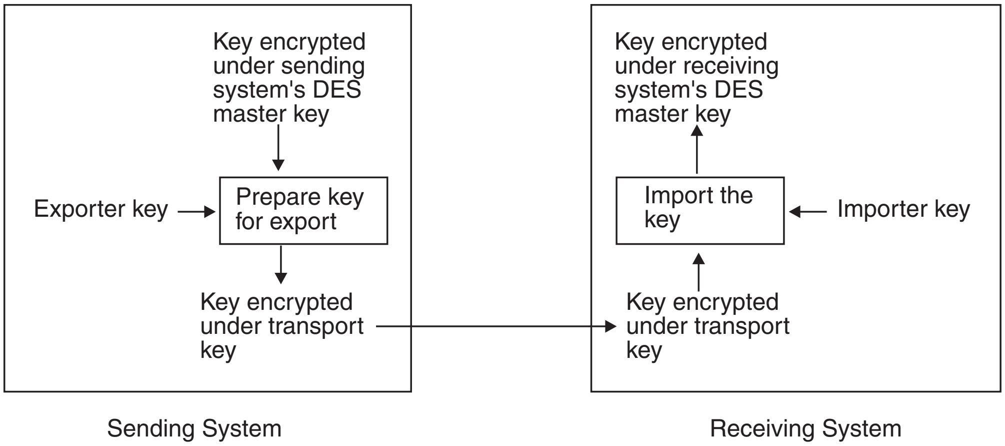

Key encrypted under sending system's DES master key
Exporter key
Prepare key for export
Key encrypted under transport key
Sending System
Key encrypted under receiving system's DES master key
Import the key
Importer key
Key encrypted under transport key
Receiving System

Figure 13. GBP PIN Verification Algorithm

VISA PIN Algorithms

The VISA PIN verification algorithm performs a multiple encipherment of a value, called the transformed security parameter (TSP), and a extraction of a 4-digit PIN verification value (PVV) from the ciphertext. The calculated PVV is compared with the referenced PVV and stored on the plastic card or data base. If they match, verification is successful.

PVV Generation Algorithm

The algorithm generates a 4-digit PIN verification value (PVV) based on the transformed security parameter (TSP).

The algorithm requires the following input parameters:

- A 64-bit TSP
- A 128-bit PVV generation key

1. A multiple encipherment of the TSP using the double-length PVV generation key is performed.
2. The ciphertext is scanned from left to right. Decimal digits are selected during the scan until four decimal digits are found. Each selected digit is placed from left to right according to the order of selection. If four decimal digits are found, those digits are the PVV.
3. If, at the end of the first scan, less than four decimal digits have been selected, a second scan is performed from left to right. During the second scan, all decimal digits are skipped and only nondecimal digits can be processed. Nondecimal digits are converted to decimal digits by subtracting 10. The process proceeds until four digits of PVV are found.

Figure 14 illustrates the PVV generation algorithm.

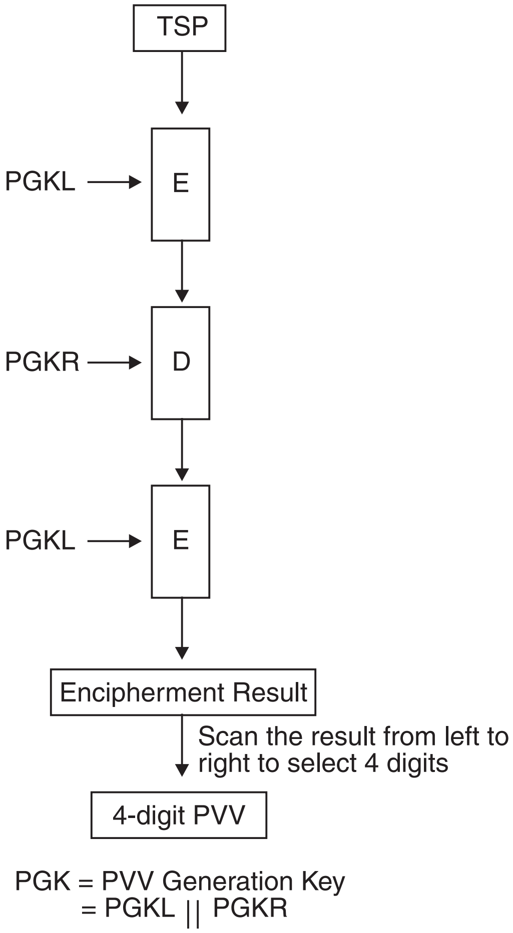


Figure 14. PVV Generation Algorithm

Programming Note: For VISA PVV algorithms, the leftmost 11 digits of the TSP are the personal account number (PAN), the leftmost 12th digit is a key table index to select the PVV generation key, and the rightmost 4 digits are the PIN. The key table index should have a value between 1 and 6, inclusive.

PVV Verification Algorithm

The algorithm requires the following input parameters:

- A 64-bit TSP

- A 16-bit referenced PVV
- A 128-bit PVV verification key

A PVV is generated using the PVV generation algorithm, except a PVV verification key rather than a PVV generation key is used. The generated PVV is compared with the referenced PVV. If they match, verification is successful.

Interbank PIN Generation Algorithm

The Interbank PIN calculation method consists of the following steps:

1. Let X denote the transaction_security parameter element converted to an array of 16 4-bit numeric values. This parameter consists of (in the following sequence) the 11 rightmost digits of the customer PAN (excluding the check digit), a constant of 6, a 1-digit key indicator, and a 3-digit validation field.
2. Encrypt X with the double-length PINGEN (or PINVER) key to get 16 hexadecimal digits (64 bits).
3. Perform decimalization on the result of the previous step by scanning the 16 hexadecimal digits from left to right, skipping any digit greater than X'9' until 4 decimal digits (for example, digits that have values from X'0' to X'9') are found.

   If all digits are scanned but 4 decimal digits are not found, repeat the scanning process, skipping all digits that are X'9' or less and selecting the digits that are greater than X'9'. Subtract 10 (X'A') from each digit selected in this scan.

   If the 4 digits that were found are all zeros, replace the 4 digits with 0100.
4. Concatenate and use the resulting digits for the Interbank PIN. The 4-digit PIN consists of the decimal digits in the sequence in which they are found.

Cipher Processing Rules

The DES defines operations on 8-byte data strings. Although the fundamental concepts of ciphering (enciphering and deciphering) and data verification are simple, there are different approaches to processing data strings that are not a multiple of 8 bytes in length. These approaches are defined in various standards and IBM products.

CBC and ANSI X3.106

ANSI standard X3.106 defines four methods of operation for ciphering. One of these modes, cipher block chaining (CBC), defines the basic method for performing ciphering on multiple 8-byte data strings. A plaintext data string, which must be a multiple of 8 bytes, is processed as a series of 8-byte groups. The ciphered result from processing an 8-byte group is exclusive ORed with the next group of 8 input bytes. The last 8-byte ciphered result is defined as an output chaining vector (OCV). ICSF stores the output chaining vector value in the *chaining_vector* parameter.

An initial chaining vector is exclusive ORed with the first group of 8 input bytes.

In summary:

- An input chaining vector (ICV) is required.
- If the *text_length* is not an exact multiple of 8 bytes, the request fails.
- The plaintext is not padded, for example, the output text length is not increased.

ANSI X9.23 and IBM 4700

An enhancement to the basic cipher block chaining mode of ANSI X3.106 is defined so the data lengths that are not an exact multiple of 8 bytes can be processed. The

ANSI X9.23 method *always* adds from 1 byte to 8 bytes to the plaintext before encipherment. The last added byte is the count of the added bytes and is in the range of X'01' to X'08'. The standard defines that the other added bytes, the pad characters, are random.

When ICSF enciphers the plaintext, the resulting ciphertext is always 1 to 8 bytes longer than the plaintext.

When ICSF deciphers the ciphertext, ICSF uses the last byte of the deciphered data as the number of bytes to be removed (the pad bytes and the count byte). The resulting plaintext is the same as the original plaintext.

The output chaining vector can be used as feedback with this method in the same way as with the X3.106 method.

In summary, for the ANSI X9.23 method:
- X9.23 processing requires the caller to supply an ICV.
- X9.23 encipher does not allow specification of a pad character.

The 4700 padding rule is similar to the X9.23 rule. The only difference is that in the X9.23 method, the padding character is not user-selected, but the padding string is selected by the encipher process.

Segmenting

The callable services can operate on large data objects. *Segmenting* is the process of dividing the function into more than one processing step. Your application can divide the process into multiple steps without changing the final outcome.

To provide segmenting capability, the MAC generation, MAC verification, and MDC generation callable services require an 18-byte system work area in the application address space that is provided as the chaining vector parameter to the callable service. The application program must not change the system work area.

Cipher Last-Block Rules

The DES defines cipher-block chaining as operating on multiples of 8 bytes. Various algorithms are used to process strings that are multiples of 8 bytes. The algorithms are generically named “last-block rules”. You select the supported last-block rules by using these keywords:
- X9.23
- IPS
- CUSP (also used with PCF)
- 4700-PAD

You specify which cipher last-block rule you want to use in the *rule_array* parameter of the callable service.

CUSP Considerations

If the length of the data to be enciphered is an exact multiple of 8 bytes, the ICV is exclusive ORed with the first 8-byte block of plaintext, and the resulting 8 bytes are passed to the DES with the specified key. The resulting 8-byte block of ciphertext is then exclusive ORed with the second 8-byte block of plaintext, and the value is enciphered. This process continues until the last 8-byte block of plaintext is to be enciphered. Because the length of this last block is exactly 8 bytes, the last block is processed in an identical manner to all the preceding blocks.

To produce the OCV, the last block of *ciphertext* is enciphered again (thus producing a double-enciphered block). The user can pass this value of the OCV as the ICV in his next encipher call to produce chaining between successive calls. The caller can alternatively pass the same ICV on every call to the callable service.

If the length of data to be enciphered is greater than 7 bytes, and is *not* an exact multiple of 8 bytes, the process is the same as that above, until the last partial block of 1 to 7 bytes is reached. To encipher the last short block, the previous 8-byte block of ciphertext is passed to the DES with the specified key. The first 1 to 7 bytes of this double-enciphered block has two uses. The first use is to exclusive OR this block with the last short block of plaintext to form the last short block of the ciphertext. The second use is to pass it back as the OCV. Thus, the OCV is the last complete 8-byte block of plaintext, doubly enciphered.

If the length of the data to be enciphered is less than 8 bytes, the ICV is enciphered under the specified key. The first 1 to 7 bytes of the enciphered ICV is exclusive ORed with the plaintext to form the ciphertext. The OCV is the enciphered ICV.

The Information Protection System (IPS)

The Information Protection System (IPS) offers two forms of chaining: block and record. Under record chaining, the OCV for each enciphered data string becomes the ICV for the next. Under block chaining, the same ICV is used for each encipherment.

Files that are enciphered directly with the ICSF encipher callable service cannot be properly deciphered using the IPS/CMS CIPHER command or the IPS/CMS subroutines. Both IPS/CMS CIPHER and AMS REPRO ENCIPHER write headers to their files that contain information (principally the ICV and chaining method) needed for decipherment. The encipher callable service does not generate these headers. Specialized techniques are described in IPS/CMS documentation to overcome some, if not all, of these limitations, depending on the chaining mode. As a rough test, you can attempt a decipherment with the CIPHER command HDWARN option, which causes CIPHER to continue processing even though the header is absent.

The encipher callable service returns an OCV used by IPS for record chaining. This allows cryptographic applications using ICSF to be compatible with IPS record chaining.

Record chaining provides a superior method of handling successive short blocks, and has better error recovery features when the caller passes successive short blocks.

The principle used by record chaining is that *the OCV is the last 8 bytes of ciphertext*. This is handled as follows:

- If the length of the data to be enciphered is an exact multiple of 8 bytes, the ICV is exclusive ORed with the first 8 byte block of plaintext, and the resulting 8 bytes are passed to the DES with the specified key. The resulting 8-byte block of ciphertext is then exclusive ORed with the second 8-byte block of plaintext, and the resulting value is enciphered. This process continues until the last 8-byte block of plaintext is to be enciphered. Because the length of this last block is exactly 8 bytes, the last block is processed in an identical manner to all the preceding blocks.

  The OCV is the last 8 bytes of ciphertext.

  The user can pass this value as the ICV in the next encipher call to produce chaining between successive calls.
- If the length of data to be enciphered is greater than 7 bytes, and is *not* an exact multiple of 8 bytes, the process is the same as that above, until the last partial

block of 1 to 7 bytes is reached. To encipher the last short block, the previous 8-byte block of ciphertext is passed to the DES with the specified key. The first 1 to 7 bytes of this doubly enciphered block is then exclusive ORed with the last short block of plaintext to form the last short block of the ciphertext. The OCV is the last 8 bytes of ciphertext.

- If the length of the data to be enciphered is less than 8 bytes, then the ICV is enciphered under the specified key. The first 1 to 7 bytes of the enciphered ICV is exclusive ORed with the plaintext to form the ciphertext. The OCV is the rightmost 8 bytes of the plaintext ICV concatenated with the short block of ciphertext. For example:

```
ICV         = ABCDEFGH
ciphertext  = XYZ
OCV         = DEFGHXYZ
```

Multiple Decipherment and Encipherment

This appendix explains multiple encipherment and decipherment and their equations.

The Integrated Cryptographic Feature uses multiple encipherment whenever it enciphers a key under a key-encrypting key like the master key or the transport key and in triple-DES encipherment for data privacy. Multiple encipherment is superior to single encipherment because multiple encipherment increases the work needed to “break” a key. ICSF provides extra protection for a key by enciphering it under an enciphering key multiple times rather than once. The multiple encipherment method for keys enciphered under a key-encrypting key uses a double-length (128 bit) key split into two 64-bit halves. Like single encipherment, multiple encipherment uses a DES based on the electronic code book (ECB) mode of encipherment.

Keys can either be double-length or single-length depending on the installation and their cryptographic function. When a single-length key is encrypted under a double-length key, multiple encipherment is performed on the key. In the multiple encipherment method, the key is encrypted under the left half of the enciphering key. The result is then decrypted under the right half of the enciphering key. Finally, this result is encrypted under the left half of the enciphering key again.

When a double-length key is encrypted with multiple encipherment, the method is similar, except ICSF uses two enciphering keys. One enciphering key encrypts each half of the double-length key. Double-length keys active on the system have two master key variants used when enciphering them.

Multiple encipherment and decipherment is not only used to protect or retrieve a cryptographic key, but they are also used to protect or retrieve 64-bit data in the area of PIN applications. For example, the following two sections use a double-length *KEK as an example to cipher a single-length key even though the same algorithms apply to cipher 64-bit data by a double-length PIN-related cryptographic key.

ICSF also supports triple-DES encipherment for data privacy using double-length and triple-length DATA keys. For this procedure the data is first enciphered using the first DATA key. The result is then deciphered using the second DATA key. This second result is then enciphered using the third DATA key when a triple-length key is provided, or reusing the first DATA key when a double-length key is provided.

Note that an asterisk (*) preceding the key means that the key is double-length. Notations in this chapter have the following meaning:

- eK(x), where x is enciphered under K
- dK(y) represents plaintext, where K is the key and y is the ciphertext

Therefore, dK(eK(x)) equals x for any 64-bit key K and any 64-bit plaintext x.

When a key (*K) to be protected is double-length, two double-length *KEKs are used. One *KEK is used for protecting the left half of the key (*K); another is for the right half. Multiple encipherment is used with the appropriate *KEK for protecting each half of the key.

Multiple Encipherment of Single-length Keys

The multiple encipherment of a single-length key (K) using a double-length *KEK is defined as follows:

```
e*KEK(K) = eKEKL(dKEKR(eKEKL(K)))
```

where KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

Figure 15 illustrates the definition.

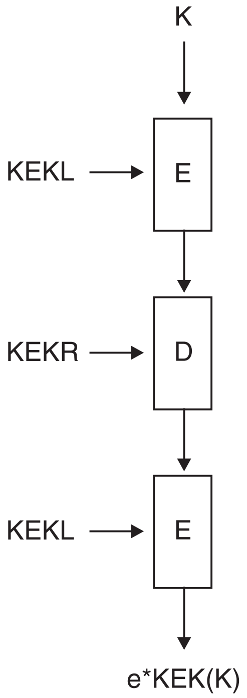


Figure 15. Multiple Encipherment of Single-length Keys

Multiple Decipherment of Single-length Keys

The multiple encipherment of an encrypted single-length key (Y = e*KEK(K)) using a double-length *KEK is defined as follows:

```
d*KEK(Y) = dKEKL(eKEKR(dKEKL(Y)))
         = d*KEK(e*KEK(K))
         = K
```

where KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

Figure 16 illustrates the definition.

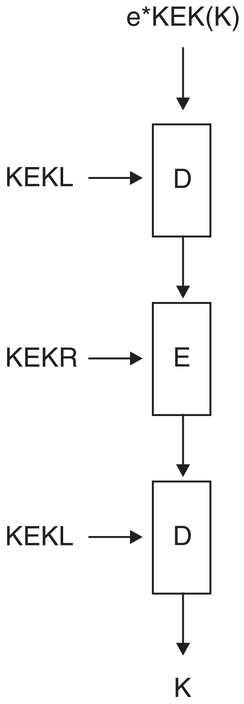


Figure 16. Multiple Decipherment of Single-length Keys

Multiple Encipherment of Double-length Keys

The multiple encipherment of a double-length key (*K) using two double-length *KEKs, *KEKa and *KEKb is defined as follows:

```
e*KEKa(KL) || e*KEKb(KR) =
              eKEKaL(dKEKaR(eKEKaL(KL))) ||
              eKEKbL(dKEKbR(eKEKbL(KR)))
```

where:

- KL is the left 64 bits of *K.
- KR is the right 64 bits of *K.
- KEKaL is the left 64 bits of *KEKa.
- KEKaR is the right 64 bits of *KEKa.
- KEKbL is the left 64 bits of *KEKb.
- KEKbR is the right 64 bits of *KEKb.
- ‖ means concatenation.

Figure 17 illustrates the definition.

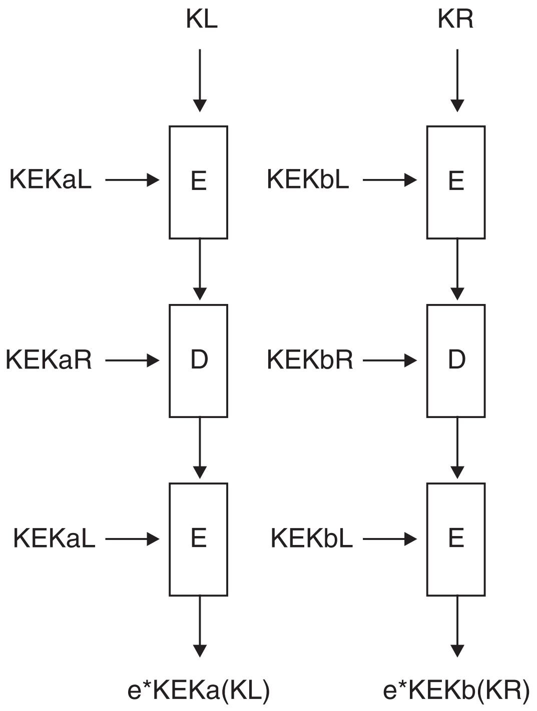


Figure 17. Multiple Encipherment of Double-length Keys

Multiple Decipherment of Double-length Keys

The multiple decipherment of an encrypted double-length key, *Y = e*KEKa(KL) || e*KEKb(KR), using two double-length *KEKs, *KEKa and *KEKb, is defined as follows:

```
D*KEKa(YL) || d*KEKb(YR)
        = dKEKaL(eKEKaR(dKEKaL(YL))) ||
          dKEKbL(eKEKbR(dKEKbL(YR)))
        = d*KEKa(e*KEKa(KL)) ||
          d*KEKb(e*KEKb(KR))
        = *K
```

where

- YL is the left 64 bits of *Y.
- YR is the right 64 bits of *Y.
- KEKaL is the left 64 bits of *KEKa.
- KEKaR is the right 64 bits of *KEKa.
- KEKbL is the left 64 bits of *KEKb.
- KEKbR is the right 64 bits of *KEKb.
- || means concatenation.

Figure 18 illustrates the definition.

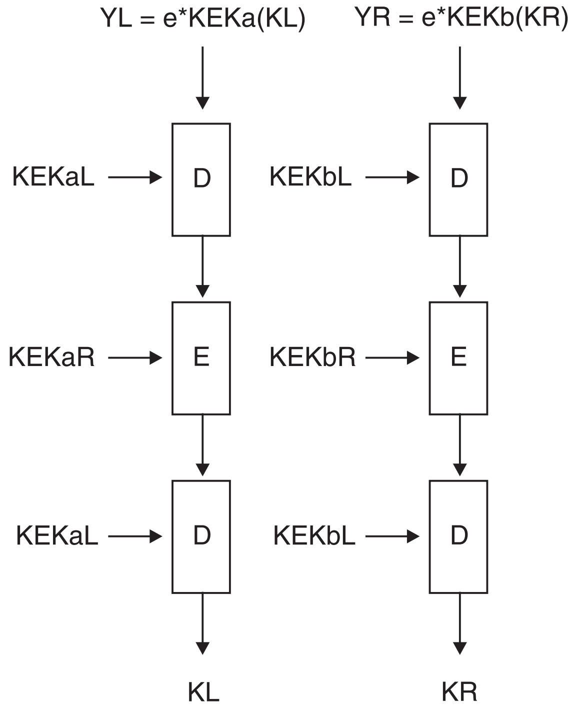


Figure 18. Multiple Decipherment of Double-length Keys

Multiple Encipherment of Triple-length Keys

The multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa and *KEKb is defined as follows:

```
e*KEKa(KL) || e*KEKb(KM) || e*KEKa(KR) =
    eKEKaL(dKEKaR(eKEKaL(KL))) ||
    eKEKbL(dKEKbR(eKEKbL(KM))) ||
    eKEKaL(dKEKaR(eKEKaL(KR)))
```

where:

- KL is the left 64 bits of **K
- KM is the next 64 bits of **K
- KR is the right 64 bits of **K
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 19 on page 438 illustrates the definition.

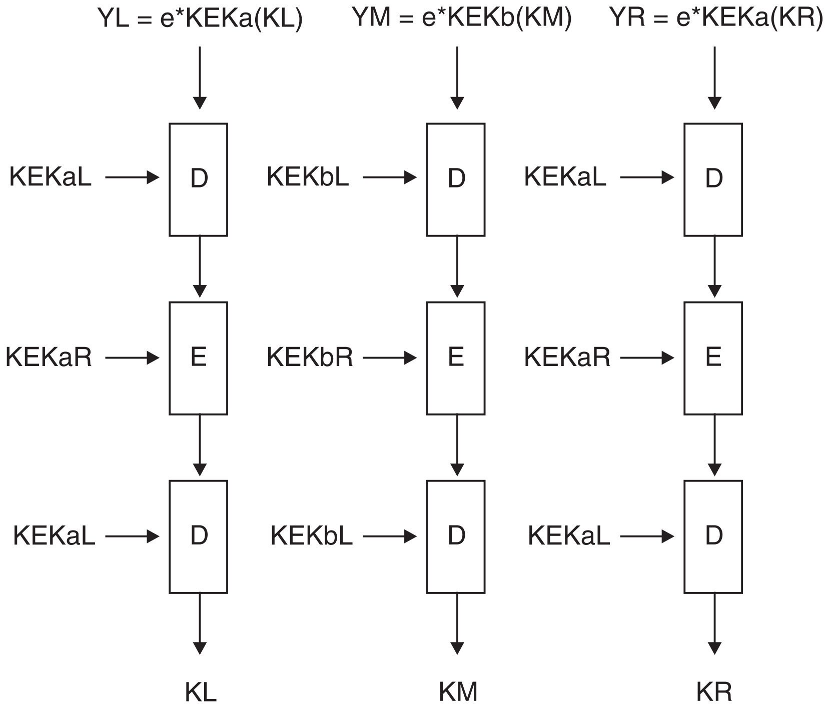


Figure 19. Multiple Encipherment of Triple-length Keys

Multiple Decipherment of Triple-length Keys

The multiple decipherment of an encrypted triple-length key **Y = e*KEKa(KL) || e*KEKb(KM) || e*KEKa(KR), using two double-length *KEKs, *KEKa and *KEKb, is defined as follows:

```
d*KEKa(YL) || d*KEKb(YM) || d*KEKa(YR)
   = dKEKaL(eKEKaR(dKEKaL(YL))) ||
     dKEKbL(eKEKbR(dKEKbL(YM))) ||
     dKEKaL(eKEKaR(dKEKaL(YR)))
   = d*KEKa(e*KEKa(KL)) ||
     d*KEKb(e*KEKb(KM)) ||
     d*KEKa(e*KEKa(KR))
   = **K
```

where:

- YL is the left 64 bits of **Y
- YM is the next 64 bits of **Y
- YR is the right 64 bits of **Y
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 20 on page 439 illustrates the definition.

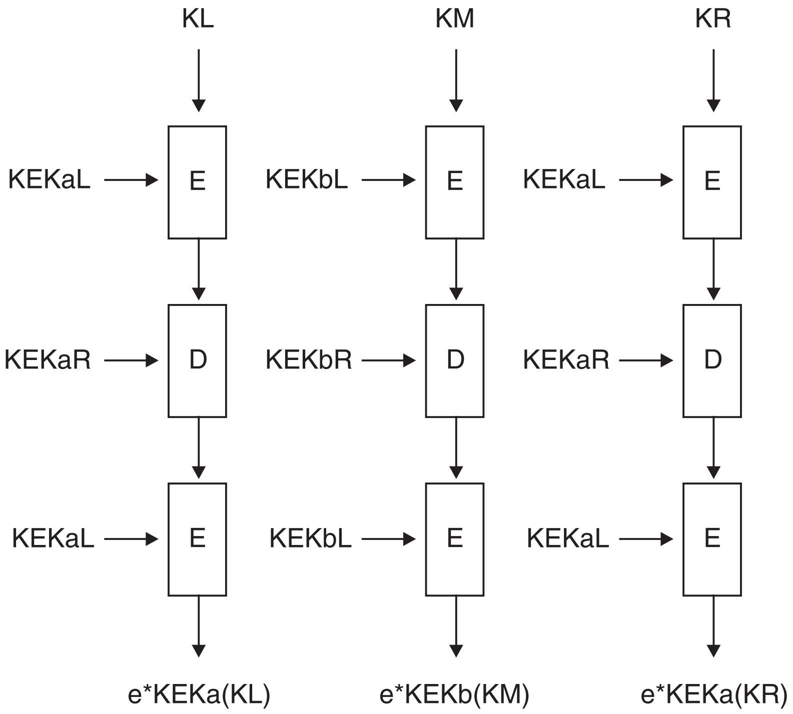


Figure 20. Multiple Decipherment of Triple-length Keys

PKA92 Key Format and Encryption Process

The PKA Symmetric Key Generate and the PKA Symmetric Key Import callable services optionally support a **PKA92** method of encrypting a DES or CDMF key with an RSA public key. This format is adapted from the IBM Transaction Security System (TSS) 4753 and 4755 product's implementation of "PKA92". The callable services do not create or accept the complete PKA92 AS key token as defined for the TSS products. Rather, the callable services only support the actual RSA-encrypted portion of a TSS PKA92 key token, the *AS External Key Block*.

Forming an External Key Block - The PKA96 implementation forms an AS External Key Block by RSA-encrypting a key block using a public key. The key block is formed by padding the key record detailed in Table 113 with zero bits on the left, high-order end of the key record. The process completes the key block with three sub-processes: masking, overwriting, and RSA encrypting.

Table 113. PKA96 Clear DES Key Record

Offset (Bytes)	Length (Bytes)	Description
Zero-bit padding to form a structure as long as the length of the public key modulus. The implementation constrains the public key modulus to a multiple of 64 bits in the range of 512 to 1024 bits. Note that government export or import regulations can impose limits on the modulus length. The maximum length is validated by a check against a value in the Function Control Vector.		
000	005	Header and flags: X'01 0000 0000'
005	016	Environment Identifier (EID), encoded in ASCII
021	008	Control vector base for the DES key

Table 113. PKA96 Clear DES Key Record (continued)

Offset (Bytes)	Length (Bytes)	Description
029	008	Repeat of the CV data at offset 021
037	008	The single-length DES key or the left half of a double-length DES key
045	008	The right half of a double-length DES key or a random number. This value is locally designated ″K.″
053	008	Random number, ″IV″
061	001	Ending byte, X'00'

Masking Sub-process - Create a mask by CBC encrypting a multiple of 8 bytes of binary zeros using K as the key and IV as the initialization vector as defined in the key record at offsets 45 and 53. Exclusive-OR the mask with the key record and call the result PKR.

Overwriting Sub-process - Set the high-order bits of PKR to B'01', and set the low-order bits to B'0110'.

Exclusive-OR K and IV and write the result at offset 45 in PKR.

Write IV at offset 53 in PKR. This causes the masked and overwritten PKR to have IV at its original position.

Encrypting Sub-process - RSA encrypt the overwritten PKR masked key record using the public key of the receiving node.

Recovering a Key from an External Key Block - Recover the encrypted DES key from an AS External Key Block by performing decrypting, validating, unmasking, and extraction sub-processes.

Decrypting Sub-process - RSA decrypt the AS External Key Block using an RSA private key and call the result of the decryption PKR. The private key must be usable for key management purposes.

Validating Sub-process - Verify that the high-order two bits of the PKR record are valued to B'01' and that the low-order four bits of the PKR record are valued to B'0110'.

Unmasking Sub-process - Set IV to the value of the 8 bytes at offset 53 of the PKR record. Note that there is a variable quantity of padding prior to offset 0. See Table 113 on page 439.

Set K to the exclusive-OR of IV and the value of the 8 bytes at offset 45 of the PKR record.

Create a mask that is equal in length to the PKR record by CBC encrypting a multiple of 8 bytes of binary zeros using K as the key and IV as the initialization vector. Exclusive-OR the mask with PKR and call the result the key record.

Copy K to offset 45 in the PKR record.

Extraction Sub-process. Confirm that:

- The four bytes at offset 1 in the key record are valued to X'0000 0000'
- The two control vector fields at offsets 21 and 29 are identical
- If the control vector is an IMPORTER or EXPORTER key class, that the EID in the key record is not the same as the EID stored in the cryptographic engine.

The control vector base of the recovered key is the value at offset 21. If the control vector base bits 40 to 42 are valued to B'010' or B'110', the key is double length. Set the right half of the received key's control vector equal to the left half and reverse bits 41 and 42 in the right half.

The recovered key is at offset 37 and is either 8 or 16 bytes long based on the control vector base bits 40 to 42. If these bits are valued to B'000', the key is single length. If these bits are valued to B'010' or B'110', the key is double length.

ANSI X9.17 Partial Notarization Method

The ANSI X9.17 notarization process can be divided into two procedures:

1. *Partial notarization*, in which the ANSI key-encrypting key (AKEK) is cryptographically combined with the origin and destination identifiers.

   Note: IBM defines this step as partial notarization. The ANSI X9.17 standard does not use the term partial notarization.
2. *Offsetting*, in which the result of the first step is exclusive-ORed with a counter value. ICSF performs the offset procedure to complete the notarization process when you use a partially notarized AKEK.

This appendix describes partial notarization for the ANSI X9.17 notarization process.

Partial Notarization

Partial notarization improves performance when you use an AKEK for many cryptographic service messages, each with a different counter value.

This section describes the steps in partial notarization. For more information about partial notarization, see “ANSI X9.17 Key Management Services” on page 27. For a description of the steps ICSF uses to complete the notarization of an AKEK or to notarize a key in one process, see *ANSI X9.17 - 1985, Financial Institution Key Management (Wholesale)*.

Notations Used in the Calculations

***KK**	The 16-byte AKEK to be partially notarized
KKL	The leftmost 8 bytes of *KK
KKR	The rightmost 8 bytes of *KK
KK	The 8-byte AKEK to be partially notarized
KK1	An 8-byte intermediate result
KK2	An 8-byte intermediate result
FMID	The 16-byte origin identifier
FMID1	The leftmost 8 bytes of FMID
FMID2	The rightmost 8 bytes of FMID
TOID	The 16-byte destination identifier
TOID1	The leftmost 8 bytes of TOID
TOID2	The rightmost 8 bytes of TOID

NSL An 8-byte intermediate result
NSL1 The leftmost 4 bytes of NSL

NSR An 8-byte intermediate result
NSR2 The rightmost 4 bytes of NSR

***KKNI** The 16-byte partially notarized AKEK
KKNIL
The leftmost 8 bytes of *KKNI
KKNIR
The rightmost 8 bytes of *KKNI
KKNI The 8-byte partially notarized AKEK

XOR Denotes the exclusive-OR operation
TOID1<<1
Denotes the ASCII TOID1 left-shifted one bit
FMID1<<1
Denotes the ASCII FMID1 left-shifted one bit
eK(X) Denotes DES encryption of plaintext X using key K
‖ Denotes the concatenation operation

Partial Notarization Calculation for a Double-Length AKEK

For a double-length AKEK, the partial notarization calculation consists of the following steps:

1. Set KK1 = KKL XOR TOID1<<1
2. Set KK2 = KKR XOR FMID1<<1
3. Set NSL = eKK2(TOID2)
4. Set NSR = eKK1(FMID2)
5. Set KKNIL = KKL XOR NSL
6. Set KKNIR = KKR XOR NSR
7. Set *KKNI = KKNIL ‖ KKNIR

Partial Notarization Calculation for a Single-Length AKEK

For a single-length AKEK, the partial notarization calculation consists of the following steps:

1. Set KK1 = KK XOR TOID1<<1
2. Set KK2 = KK XOR FMID1<<1
3. Set NSL = eKK2(TOID2)
4. Set NSR = eKK1(FMID2)
5. Set NSL = NSL1 ‖ NSR2
6. Set KKNI = KK XOR NSL

Transform CDMF Key Algorithm

The CDMF key transformation algorithm uses a 64-bit cryptographic key.

1. Set parity bits of the key to zero by ANDing the key with X'FEFEFEFEFEFEFEFE' to produce Kx.
2. Using DES, encipher Kx under the constant K1.
3. XOR this value with Kx to produce Ky.
4. AND Ky with X'0EFE0EFE0EFE0EFE' to produce Kz.
5. Using DES, encipher Kz under K2 to produce eK2(Kz).
6. Adjust eK2(Kz) to odd parity in each byte. The result is the transformed key.

The following figure illustrates these steps. (e indicates DES encryption.)

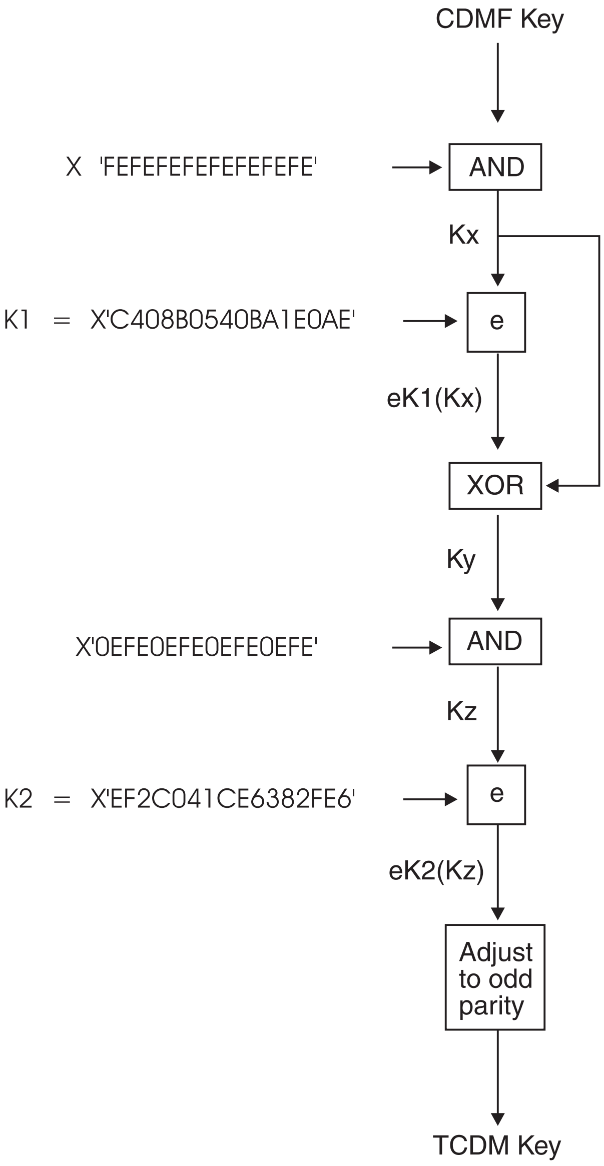


Figure 21. The CDMF Key Transformation Algorithm

Appendix G. EBCDIC and ASCII Default Conversion Tables

This section presents tables showing EBCDIC to ASCII and ASCII to EBCDIC conversion tables. In the table headers, EBC refers to EBCDIC and ASC refers to ASCII.

Table 114 shows the EBCDIC to ASCII default conversion table.

Table 114. EBCDIC to ASCII Default Conversion Table

EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC
00	00	20	81	40	20	60	2D	80	F8	A0	C8	C0	7B	E0	5C
01	01	21	82	41	A6	61	2F	81	61	A1	7E	C1	41	E1	E7
02	02	22	1C	42	E1	62	DF	82	62	A2	73	C2	42	E2	53
03	03	23	84	43	80	63	DC	83	63	A3	74	C3	43	E3	54
04	CF	24	86	44	EB	64	9A	84	64	A4	75	C4	44	E4	55
05	09	25	0A	45	90	65	DD	85	65	A5	76	C5	45	E5	56
06	D3	26	17	46	9F	66	DE	86	66	A6	77	C6	46	E6	57
07	7F	27	1B	47	E2	67	98	87	67	A7	78	C7	47	E7	58
08	D4	28	89	48	AB	68	9D	88	68	A8	79	C8	48	E8	59
09	D5	29	91	49	8B	69	AC	89	69	A9	7A	C9	49	E9	5A
0A	C3	2A	92	4A	9B	6A	BA	8A	96	AA	EF	CA	CB	EA	A0
0B	0B	2B	95	4B	2E	6B	2C	8B	A4	AB	C0	CB	CA	EB	85
0C	0C	2C	A2	4C	3C	6C	25	8C	F3	AC	DA	CC	BE	EC	8E
0D	0D	2D	05	4D	28	6D	5F	8D	AF	AD	5B	CD	E8	ED	E9
0E	0E	2E	06	4E	2B	6E	3E	8E	AE	AE	F2	CE	EC	EE	E4
0F	0F	2F	07	4F	7C	6F	3F	8F	C5	AF	F9	CF	ED	EF	D1
10	10	30	E0	50	26	70	D7	90	8C	B0	B5	D0	7D	F0	30
11	11	31	EE	51	A9	71	88	91	6A	B1	B6	D1	4A	F1	31
12	12	32	16	52	AA	72	94	92	6B	B2	FD	D2	4B	F2	32
13	13	33	E5	53	9C	73	B0	93	6C	B3	B7	D3	4C	F3	33
14	C7	34	D0	54	DB	74	B1	94	6D	B4	B8	D4	4D	F4	34
15	B4	35	1E	55	A5	75	B2	95	6E	B5	B9	D5	4E	F5	35
16	08	36	EA	56	99	76	FC	96	6F	B6	E6	D6	4F	F6	36
17	C9	37	04	57	E3	77	D6	97	70	B7	BB	D7	50	F7	37
18	18	38	8A	58	A8	78	FB	98	71	B8	BC	D8	51	F8	38
19	19	39	F6	59	9E	79	60	99	72	B9	BD	D9	52	F9	39
1A	CC	3A	C6	5A	21	7A	3A	9A	97	BA	8D	DA	A1	FA	B3
1B	CD	3B	C2	5B	24	7B	23	9B	87	BB	D9	DB	AD	FB	F7
1C	83	3C	14	5C	2A	7C	40	9C	CE	BC	BF	DC	F5	FC	F0
1D	1D	3D	15	5D	29	7D	27	9D	93	BD	5D	DD	F4	FD	FA
1E	D2	3E	C1	5E	3B	7E	3D	9E	F1	BE	D8	DE	A3	FE	A7
1F	1F	3F	1A	5F	5E	7F	22	9F	FE	BF	C4	DF	8F	FF	FF

Table 115 shows the ASCII to EBCDIC default conversion table.

Table 115. ASCII to EBCDIC Default Conversion Table

ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC	ASC	EBC
00	00	20	40	40	7C	60	79	80	43	A0	EA	C0	AB	E0	30
01	01	21	5A	41	C1	61	81	81	20	A1	DA	C1	3E	E1	42
02	02	22	7F	42	C2	62	82	82	21	A2	2C	C2	3B	E2	47
03	03	23	7B	43	C3	63	83	83	1C	A3	DE	C3	0A	E3	57
04	37	24	5B	44	C4	64	84	84	23	A4	8B	C4	BF	E4	EE
05	2D	25	6C	45	C5	65	85	85	EB	A5	55	C5	8F	E5	33
06	2E	26	50	46	C6	66	86	86	24	A6	41	C6	3A	E6	B6
07	2F	27	7D	47	C7	67	87	87	9B	A7	FE	C7	14	E7	E1
08	16	28	4D	48	C8	68	88	88	71	A8	58	C8	A0	E8	CD
09	05	29	5D	49	C9	69	89	89	28	A9	51	C9	17	E9	ED
0A	25	2A	5C	4A	D1	6A	91	8A	38	AA	52	CA	CB	EA	36
0B	0B	2B	4E	4B	D2	6B	92	8B	49	AB	48	CB	CA	EB	44
0C	0C	2C	6B	4C	D3	6C	93	8C	90	AC	69	CC	1A	EC	CE
0D	0D	2D	60	4D	D4	6D	94	8D	BA	AD	DB	CD	1B	ED	CF
0E	0E	2E	4B	4E	D5	6E	95	8E	EC	AE	8E	CE	9C	EE	31
0F	0F	2F	61	4F	D6	6F	96	8F	DF	AF	8D	CF	04	EF	AA
10	10	30	F0	50	D7	70	97	90	45	B0	73	D0	34	F0	FC
11	11	31	F1	51	D8	71	98	91	29	B1	74	D1	EF	F1	9E
12	12	32	F2	52	D9	72	99	92	2A	B2	75	D2	1E	F2	AE
13	13	33	F3	53	E2	73	A2	93	9D	B3	FA	D3	06	F3	8C
14	3C	34	F4	54	E3	74	A3	94	72	B4	15	D4	08	F4	DD
15	3D	35	F5	55	E4	75	A4	95	2B	B5	B0	D5	09	F5	DC
16	32	36	F6	56	E5	76	A5	96	8A	B6	B1	D6	77	F6	39
17	26	37	F7	57	E6	77	A6	97	9A	B7	B3	D7	70	F7	FB
18	18	38	F8	58	E7	78	A7	98	67	B8	B4	D8	BE	F8	80
19	19	39	F9	59	E8	79	A8	99	56	B9	B5	D9	BB	F9	AF
1A	3F	3A	7A	5A	E9	7A	A9	9A	64	BA	6A	DA	AC	FA	FD
1B	27	3B	5E	5B	AD	7B	C0	9B	4A	BB	B7	DB	54	FB	78
1C	22	3C	4C	5C	E0	7C	4F	9C	53	BC	B8	DC	63	FC	76
1D	1D	3D	7E	5D	BD	7D	D0	9D	68	BD	B9	DD	65	FD	B2
1E	35	3E	6E	5E	5F	7E	A1	9E	59	BE	CC	DE	66	FE	9F
1F	1F	3F	6F	5F	6D	7F	07	9F	46	BF	BC	DF	62	FF	FF

Appendix H. Access Control Points and Callable Services

Access to services that are executed on the PCI Cryptographic Coprocessor is through Access Control Points in the DEFAULT Role. To execute callable services on the PCI Cryptographic Coprocessor, access control points must be enabled for each service in the DEFAULT Role. The ability to enable/disable access control points in the DEFAULT Role was introduced on OS/390 V2R10 through APAR OW46381 for the Trusted Key Entry Workstation. For systems that do not use the optional TKE Workstation, all access control points (current and new) are enabled in the DEFAULT Role with the appropriate microcode level on the PCI Cryptographic Coprocessor. New TKE customers and Non-TKE customers have all access control points enabled. This is also true for brand new TKE V3.1 users (not converting from TKE V3.0).

Note: Access control point DKYGENKY-DALL is always disabled in the DEFAULT Role for all customers (TKE and Non-TKE). A TKE Workstation is required to enable this access control point for the Diversified Key Generate service.

For existing TKE V3.0 users, upgrading to TKE V3.1 (APAR OW46381 and its corresponding ECA), current access control points in the DEFAULT Role are enabled. Any new access control points are disabled in the DEFAULT Role and must be enabled through TKE if the service is required.

Notes:

1. APAR OW46381 will update the TKE Host Code
2. ECA 186 will update the TKE Workstation Code
3. The latest or most current driver is required for the PCI Cryptographic Coprocessor microcode for the S/390 G5 Enterprise Server or the S/390 G6 Enterprise Server
4. The latest or most current driver is required for the PCI Cryptographic Coprocessor microcode for the IBM @server zSeries 900

All of the above components are required for complete access control point support.

Access to services which execute on the Cryptographic Coprocessor Feature is through SAF. Disablement through SAF is sufficient to prevent execution of a service by either the Cryptographic Coprocessor Feature or the PCI Cryptographic Coprocessor. For functions which can be executed on the PCI Cryptographic Coprocessor, enablement of the function requires that the function be enabled through SAF and through the access control point in the DEFAULT Role.

If you are on OS/390 V2 R10, using a TKE V3.0 workstation, access control points for new services (requiring APARs OW46380 and OW46382) will be disabled. Existing access control points will be enabled in the DEFAULT Role. APAR OW46381 must be installed to enable the OS/390 V2 R10 interface. This will allow the TKE Administrator to enable any new access control points for ICSF services that execute in the PCI Cryptographic Coprocessor under the DEFAULT Role.

Access Control Points (requiring APARs OW46380 and OW46382) for OS/390 V2R10 are:

- DATAM Key Management Control

Note: For existing TKE installations (upgrading to TKE V3.1), it is required that this access control point be enabled. Failure to do so will result in processing errors for Double MAC keys in Key Import, Key Export, and Key Generate.

- Diversified Key Generate - Single length or same halves
- Diversified Key Generate - CLR8-ENC
- Diversified Key Generate - TDES-ENC
- Diversified Key Generate - TDES-DEC
- Diversified Key Generate - SESS-XOR
- Diversified Key Generate - DKYGENKY-DALL

  Note: This access control point is always disabled in the DEFAULT Role for all customers (TKE and Non-TKE). A TKE Workstation is required to enable the function.
- MAC Generate - For existing TKE installations, it is recommended that this access control point be enabled.
- MAC Verify - For existing TKE installations, it is recommended that this access control point be enabled.

Access Control Points for z/OS V1 R2 are:

- PKA Key Token Change
- Secure Messaging for Keys
- Secure Messaging for PINs

| Access Control Points for z/OS V1 R3 are:

| - UKPT - PIN Verify, PIN Translate

If an access control point is disabled, the corresponding ICSF callable service will fail during execution with an access denied error.

Access Control Point	Callable Service
Clear PIN Encrypt	CSNBCPE
Clear PIN Generate - 3624	CSNBPGN
Clear PIN Generate - GPB	CSNBPGN
Clear PIN Generate - VISA PVV	CSNBPGN
Clear PIN Generate - Interbank	CSNBPGN
Clear Pin Generate Alternate - 3624 Offset	CSNBCPA
Clear PIN Generate Alternate - VISA PVV	CSNBCPA
Control Vector Translate	CSNBCVT
Cryptographic Variable Encipher	CSNBCVE
CVV Generate	CSNBCSG
CVV Verify	CSNBCSV
DATAM Key Management Control	CSNBKGN, CSNBKIM, CSNBKEX and CSNBDKG
Data Key Export	CSNBDKX
Data Key Import	CSNBDKM
Digital Signature Generate	CSNDDSG
Diversified Key Generate - CLR8–ENC	CSNBDKG

Diversified Key Generate - TDES-ENC	CSNBDKG
Diversified Key Generate - TDES-DEC	CSNBDKG
Diversified Key Generate - SESS-XOR	CSNBDKG
Diversified Key Generate - single length or same halves	CSNBDKG
DKYGENKY - DALL	CSNBDKG
Encrypted PIN Generate - 3624	CSNBEPG
Encrypted PIN Generate - GPB	CSNBEPG
Encrypted PIN Generate - Interbank	CSNBEPG
Encrypted PIN Translate - Translate	CSNBPTR
Encrypted PIN Translate - Reformat	CSNBPTR
Encrypted PIN Verify - 3624	CSNBPVR
Encrypted PIN Verify - GPB	CSNBPVR
Encrypted PIN Verify - VISA PVV	CSNBPVR
Encrypted PIN Verify - Interbank	CSNBPVR
Key Export	CSNBKEX
Key Generate - OPIM, OPEX, IMEX, etc.	CSNBKGN
Key Generate - EX, IM, OP	CSNBKGN
Key Generate - CVARs	CSNBKGN
Key Generate - SINGLE-R	CSNBKGN
Key Import	CSNBKIM
Key Part Import - first key part	CSNBKPI
Key Part Import - middle and final	CSNBKPI
Key Translate	CSNBKTR
MAC Generate	CSNBMGN
MAC Verify	CSNBMVR
PKA Decrypt	CSNDPKD
PKA Encrypt	CSNDPKE
PKA Key Generate	CSNDPKG
PKA Key Generate - Clear	CSNDPKG
PKA Key Generate - Clone	CSNDKPG
PKA Key Import	CSNDPKI
PKA Key Token Change	CSNDKTC
Prohibit Export	CSNBPEX
Retained Key Delete	CSNDRKD
Retained Key List	CSNDRKL
Secure Key Import - IM	CSNBSKI or CSNBSKM
Secure Key Import - OP	CSNBSKI or CSNBSKM
Secure Messaging for Keys	CSNBSKY
Secure Messaging for PINs	CSNBSPN
SET Block Compose	CSNDSBC
SET Block Decompose	CSNDSBD

SET Block Decompose - PIN ext IPINENC	CSNDSBD
SET Block Decompose - PIN ext OPINENC	CSNDSBD
Symmetric Key Export - PKCS-1.2	CSNDSYX
Symmetric Key Export - ZERO-PAD	CSNDSYX
Symmetric Key Generate - PKA92	CSNDSYG
Symmetric Key Generate - PKCS-1.2	CSNDSYG
Symmetric Key Generate - ZERO-PAD	CSNDSYG
Symmetric Key Import - PKA92 KEK	CSNDSYI
Symmetric Key Import - PKA92 PIN Key	CSNDSYI
Symmetric Key Import - PKCS-1.2	CSNDSYI
Symmetric Key Import - ZERO-PAD	CSNDSYI
UKPT - PIN Verify, PIN Translate	CSNBPVR and CSNBPTR

Notes:

1. To use PKA Key Generate - Clear or PKA Key Generate - Clone, the PKA Key Generate access control point must be enabled or the callable service will fail.
2. To use SET Block Decompose - PIN ext IPINENC or PIN ext OPINENC, the SET Block Decompose access control point must be enabled or the callable service will fail.
3. Diversified Key Generate - single length or same halves requires either Diversified Key Generate - TDES-ENC or Diversified Key Generate - TDES-DEC be enabled.

Appendix I. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen-readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen-readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using it to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Volume I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Notices

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS Integrated Cryptographic Service Facility.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
AS/400
CICS
ES/3090
ES/9000
IBM
IBMLink
Multiprise
MVS
MVS/ESA
MVS/SP
OS/390
Parallel Sysplex
Personal Security
Processor Resource/Systems Manager
PR/SM
RACF
Resource Link
RMF
S/370
S/390
S/390 Parallel Enterprise Server
System/390
VTAM
3090

zSeries
z/OS

The e-business logo is a trademark of IBM.

The following terms are trademarks of other companies:

BSAFE	RSA Data Security, Incorporated
MasterCard	MasterCard International, Incorporated
Netscape	Netscape Communications Corporation
SET	SET Secure Electronic Transaction, LLC
UNIX	The Open Group
VISA	VISA International Service Association

Other company, product, and service names may be trademarks or service marks of others.

Index

Readers' Comments — We'd Like to Hear from You

z/OS
Integrated Cryptographic Service Facility
Application Programmer's Guide

Publication No. SA22-7522-02

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Company or Organization

Phone No.

Address

Readers' Comments — We'd Like to Hear from You
SA22-7522-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY
12601-5400

Fold and Tape **Please do not staple** Fold and Tape

SA22-7522-02

Cut or Fold
Along Line

Program Number: 5694-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SA22-7522-02